

AB

An Interpretation of Standard ML in Type Theory

Robert Harper Christopher Stone

June 27 1997

CMU-CS-97-147

School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

This report also appears as Fox Memorandum CMU-CS-FOX-97-01

19980106 048

This research was sponsored in part by the Advanced Research Projects Agency CSTD under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050, in part by the National Science Foundation under Grant No. CCR-9502674, and in part by the US Army Research Office under Grant No. DAAH04-94-G-0289. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Advanced Research Projects Agency, the U.S. Government or the National Science Foundation.

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

DTIC QUALITY INSPECTED 4

Keywords: Standard ML Γ type theory Γ semantics Γ programming language design Γ compilation Γ translation Γ elaboration Γ static semantics Γ dynamic semantics

Abstract

We define an interpretation of Standard ML into type theory. The interpretation takes the form of a set of elaboration rules reminiscent of the static semantics given in *The Definition of Standard ML*. In particular, the elaboration rules are given in a relational style, exploiting indeterminacy to avoid over-commitment to specific implementation techniques. Elaboration consists of identifier scope resolution, type checking and type inference, expansion of derived forms, pattern compilation, overloading resolution, equality compilation, and the coercive aspects of signature matching.

The underlying type theory is an explicitly-typed λ -calculus with product, sum, function, and recursive types, together with module types derived from the translucent sum formalism of Harper and Lillibridge. Programs of the type theory are given a type-passing dynamic semantics compatible with constructs such as polymorphic equality that rely on type analysis at run-time.

This document supercedes the previous CMU CS technical reports CMU-CS-96-136 and CMU-CS-96-136R. The revision reflects our experience in implementing the specified elaborator, and includes several essential corrections and simplifications to the presentation.

Contents

1	Introduction	3
1.1	Overview	3
1.2	Notes on Implementation	4
1.3	Major Differences from Standard ML	6
1.3.1	Value Restriction	6
1.3.2	Abstype	6
1.3.3	Local and Higher-Order Functors	6
1.3.4	Top-level	7
1.4	Major Technical Differences from Version 2	7
1.4.1	Internal Language	7
1.4.2	Datatypes	7
1.4.3	Generativity	7
1.4.4	Subtyping	8
1.4.5	Syntactic concatenation	8
2	Internal Language Abstract Syntax	9
2.1	Constructors and Kinds	9
2.2	Expressions	10
2.3	Modules and Signatures	11
2.4	Useful Abbreviations and Notation	11
2.5	Bindings and Scope	14
3	Static Semantics	16
3.1	Introduction	16
3.2	Notation	16
3.3	Judgment Forms	17
3.4	Inference Rules	18
3.4.1	Well-formed Declarations	18
3.4.2	Well-formed Bindings	18
3.4.3	Well-formed Kinds	19
3.4.4	Well-formed Constructors	19
3.4.5	Constructor Equivalence	20
3.4.6	Well-formed Expressions	21
3.4.7	Well-formed Signatures	24
3.4.8	Signature Subtyping	24
3.4.9	Signature Equivalence	25
3.4.10	Well-formed Modules	26
3.4.11	Valuability	27
4	Dynamic Semantics	29
4.1	Introduction	29
4.2	Transition Relation	30
4.2.1	Search for Next Redex	30

4.2.2	Reduction	32
5	External Language	34
5.1	Notation	34
5.2	Grammar of the Abstract Syntax	34
5.3	Syntactic Restrictions	36
6	Elaboration	37
6.1	Introduction	37
6.2	Notation	38
6.3	Initial Basis	39
6.4	Derived Forms	39
6.5	Judgment Forms	41
6.6	Translation Rules	41
6.6.1	Expressions	41
6.6.2	Matches	44
6.6.3	Declarations	45
6.6.4	Structure Expressions	47
6.6.5	Structure Bindings	48
6.6.6	Functor Bindings	49
6.6.7	Specifications	49
6.6.8	Signature Expressions	51
6.6.9	Type Expressions	51
6.6.10	Type Definitions	52
6.6.11	Type Descriptions	52
6.6.12	Equality Type Descriptions	52
6.6.13	Datatype Definitions	52
6.7	Polymorphic Instantiation	54
6.8	Equality Compilation	54
6.9	Pattern Compilation	56
6.10	Coercion Compilation	59
6.11	Signature Patching	61
6.12	Lookup Rules	62
6.12.1	Context Lookup	62
6.12.2	Signature Lookup	63
6.13	Overloading	64
7	Properties	64
7.1	Properties of the Internal Language	64
7.1.1	Static Semantics	64
7.1.2	Dynamic Semantics	67
7.1.3	Soundness of the Internal Language	68
7.2	Properties of the Elaborator	70
8	Acknowledgments	72

1 Introduction

1.1 Overview

This document consists of a type-theoretic account of a variant of (Revised) Standard ML [MTHM97] hereafter referred to simply as Standard ML or SML. The approach taken here is to elaborate SML abstract syntax (the *external language* or EL) into an explicitly-typed λ -calculus (the *internal language* or IL). The internal language is designed to be as simple and orthogonal as possible while still being able to represent the entire Standard ML language.

The translation is presented by a set of inference rules reminiscent of the static semantics given in *The Definition of Standard ML* with the internal language playing the role of the static semantic objects of *The Definition*. The translation rules typically define the translation of a phrase in terms of the translation of its constituent phrases subject to context-sensitive constraints expressed by the internal language type system. Context-sensitive formation constraints are expressed by type checking constraints on the translation. Type propagation is controlled by a combination of the translucent sum formalism together with the representation of abstract types as modules with opaque type components. The internal language ensures that abstraction is respected and moreover provides the requisite association of the abstract type to its representation at run-time.

We believe that looking at Standard ML language features in terms of how they translate into type theory clarifies a number of issues in the design of SML. For example the notion of “type generativity” is replaced by explicit control of type propagation.

The internal language may be shared among many different language descriptions given in the style presented here. In particular we envision the possibility of using the IL given here (with very minor modifications) for Scheme, Caml Special Light or Haskell. A more long-term goal would be to see what changes would be needed—especially with regards to object-oriented features—in order to handle Java or Modula-3.

The elaboration has application to language implementation as well as it may be viewed as a reference implementation of a front-end for an SML compiler. In particular we are using this translation as a guide to our re-implementation of the TIL/ML compiler [HM95, Mor95, Tar96, TMC⁺96]. Compilers for other languages defined by interpretation into the internal language could share the back end of the TIL/ML compiler; only a front-end need be written for each specific language.

There are some disadvantages to our approach. The translation of some SML language features is quite complex. To some extent this is a direct reflection of the intrinsic complexity of mechanisms such as datatype definitions (which introduce several mutually recursive abstract types each a multinary sum of multinary product types) and polymorphic recursive function bindings. However ensuring that the invariants of our encodings are respected everywhere in the document can be tedious and error-prone.

Furthermore understanding the elaboration requires understanding the internal language as well though this may be mitigated if the same internal language can be used for several different external languages.

The type-theoretic interpretation of Standard ML is divided into three main parts.

Type Structure of the Internal Language. The internal language is an explicitly-typed λ -calculus with a second-class modules system.

The core of the internal language is based on the XML and λ^{ML} calculi [HM93, HMM90]. The constructors of kind Ω (where Ω is the kind of types) include partial and total function types, record types, sum types, reference types, recursive types, and a single extensible sum. Additionally, the constructors are extended with (restricted) tuples of constructors and functions at the constructor level. Note that there are no polymorphic types (polytypes) in our system.

The modules system is based on the translucent sum or manifest type modules calculi [HL94, Ler94]. In addition to translucent signatures, we have total and partial functor signatures. Our subtyping relation on signatures involves only forgetting of type definitions and totality, and not dropping or reordering components. This means that subtyping has no run-time effect.

The two levels are connected by the ability to define modules local to a core expression.

Dynamic Semantics of the Internal Language. The dynamic semantics is essentially a contextual semantics [WF91]. The handling of references and exceptions is similar to Harper's account of polymorphic references [Har93].

Elaboration. The translation is defined by a series of translation judgments of the general form

$$\Gamma \vdash EL\text{-}phrase \rightsquigarrow IL\text{-}phrase : IL\text{-}classifier$$

where Γ is an internal-language context extended with information mapping EL identifiers to IL variables.

1.2 Notes on Implementation

This document is intended as an experiment in formally specifying the first stage of the TIL compiler, and additionally to see if this generates a plausible definition for a language. We are not advocating that a compiler should generate exactly the IL code given here, nor even that the internal language of the compiler should be exactly the IL we present. However, any implementation of an elaborator should be essentially *equivalent* to what we present, and we suggest the compiler's internal language should be *definable* in terms of the IL we present here. Differences we would expect in an actual implementation include:

- The internal language is likely to be extended to make our derived forms into primitives, possibly along with other definable forms.
- A compiler is likely to use a much more complicated pattern compiler, generating much more efficient pattern-matching code (in particular, making better use of cases).
- Clausal function definitions would not be sequenced through uses of the exception mechanism.

- A compiler is likely to use a much more efficient implementation of elaboration contexts Γ avoiding the the sequential search which we have formalized.
- Equality functions would be cached Γ so that multiple equality tests for values of the same type would not generate multiple equality functions.
- In order to achieve SML's typing behavior Γ locally-defined types at the module level are hidden through renaming rather than through use of internal-language local bindings. For simplicity Γ all module-level local definitions (including values) are translated in this fashion Γ though this is not required Γ and could cause wasted space at run-time. An implementation would use a more sophisticated criterion for which local bindings to rename.

1.3 Major Differences from Standard ML

1.3.1 Value Restriction

The value restriction used in this translation is different than that specified in SML.

- We treat

$$\text{val } \langle \text{rec} \rangle \text{ } pat_1 = expr_1 \text{ and } \dots \text{ and } pat_n = expr_n$$

as a derived form for

$$\text{val } \langle \text{rec} \rangle (pat_1, \dots, pat_n) = (expr_1, \dots, expr_n).$$

- When v is a value Γ we treat the binding $\text{val } h :: t = v$ as completely equivalent to $\text{val } h = \text{hd } v$ and $\text{val } t = \text{tl } v$. Since in the latter cases h and t will not be generalized (hd and tl may raise exceptions!) Γ h and t are not made polymorphic in the former case.

The translation allows variables in a `val` binding to be generalized if the translations of the bindings are “valuable.” At the external language level Γ this is equivalent to the right-hand side being a value Γ and the variable not be in the argument of a constructor pattern Γ unless the constructor is for a single-constructor datatype. Note that tuple-patterns pose no problem Γ since projection is a total function.

This corresponds to the notion of “polymorphism as substitution” Γ in which

$$\text{let val } id = expr \text{ in } expr' \text{ end}$$

is considered equivalent to $\{expr/id\}expr'$ when $expr$ is a value.

Because of this difference Γ a few variables that would have been polymorphic according to the SML specification will not be generalized in this treatment.

1.3.2 Abstype

We do not consider `abstype` declarations in this document because abstract data types can be easily created using the SML module system instead. There would be no technical difficulties in including `abstype` Γ if desired.

1.3.3 Local and Higher-Order Functors

Our external language permits structure and functor declarations within a `let` or `local` declarations Γ and also within `structure` and `functor` declarations; this is a reflection of the fact that the internal language has local and higher-order modules.

These higher-order modules are based on the translucent sum formalism [HL94]. As this is compatible with first-class modules (which are not included in this definition) Γ our higher-order functors propagate less type-sharing information than those in the system of MacQueen and Tofte [MT94]. Finding a clean type-theoretic description of this latter system is the subject of current research.

1.3.4 Top-level

Since we allow locally-defined modules in our external language it suffices to consider programs to be closed expressions of type *ans* where *ans* is a fixed base type of answers. We do not explicitly consider the translation of SML top-level declarations.

Note that this does not require an *implementation* to accept only complete programs of some fixed type. For example a batch compiler may expect a sequence of structure functors and signature bindings (SML top-level declarations possibly separately compiled) which are then “logically” treated as a sequence of declarations in a wrapper generated by the compiler. For a UNIX system for example the wrapper might be along the lines of:

```
let ... in 0 end handle _ => ~1
```

where *ans* is chosen to be the type *int* and a program returns either 0 for normal termination or ~1 for an uncaught exception.

To enable this interpretation we view **signature** declarations as simple “macros” defining abbreviations for signatures. The translation assumes abbreviations have previously been expanded out in the program body.

1.4 Major Technical Differences from Version 2

1.4.1 Internal Language

Some minor changes have been made to the internal language for purposes of efficient implementation (e.g. the unbundling of mutual recursion and projection operations at the constructor and expression levels) or for the purposes of a cleaner elaboration or internal language (e.g. the generalization of the kind **structure** and the replacement of partial destructuring operations with total versions at the expression level).

1.4.2 Datatypes

The rule for datatypes has been simplified; additionally the internal language has been extended to allow all valid SML datatypes to elaborate. However certain datatypes that SML defines as admitting equality would require polymorphic recursion to implement this equality function. Since our language does not currently admit polymorphic recursion (which would correspond to recursive functors) these datatypes do not admit equality in our formulation.

1.4.3 Generativity

A technical problem with our presentation caused us to reject some legal Standard ML programs because of type information was not tracked properly. This problem involved abstract types defined in anonymous structure expressions defined locally in **let** or **local** at the module level or hidden by SML’s transparent ascription (**strexpr** : **sigexp**). This has been corrected by:

1. Restricting the EL to named form at the module level following Leroy [Ler96]. This can always be achieved by a simple prepass over the program. (The grammar in Section 5 shows what we mean by named form.)

2. In the case that `let` and `local` module forms define abstract types locally Γ in the translation we augment the bodies of these forms with extra “hidden” fields containing these abstract types. In conjunction with the restriction to named form Γ this has the effect of causing types concealed by transparent ascription to be *renamed* rather than dropped entirely.¹

Although the internal language module system presented here does not have most-specific signatures in general Γ we also ensure that all modules generated by the elaborator do have most-specific signatures. This means that the elaborator never requires “guessing” of signatures for modules such as functor applications.

1.4.4 Subtyping

We have dropped the implicit subsumption between total and partial function types; every expression now has a unique most-specific types up to equivalence. Unfortunately this means datatype and exception constructors must now be handled specially when used as function values. However Γ a full subtyping relation would seem to be overkill—at least until features such as objects are added to the internal language.

1.4.5 Syntactic concatenation

Rather than using the starred structure convention for general concatenation of modules Γ we use a syntactic concatenation of structure field bindings. This includes renaming to prevent ill-formed constructions where two fields share the same label.

¹For simplicity, the rules essentially augment the bodies of `let` and `local` with *all* the locally-defined quantities. Most of these—including all value components and transparent type components—are obviously unnecessary and undesirable in an actual implementation.

2 Internal Language Abstract Syntax

2.1 Constructors and Kinds

$con ::=$	var	type variables
	$Int \mid Float \mid Char \mid \dots$	base types
	$\{rdecs\}$	record type
	$con \text{ Ref}$	reference type
	$con \rightarrow con'$	partial function type
	$con \twoheadrightarrow con'$	total function type
	$Tagged$	extensible sum type
	$con \text{ Tag}$	exception-tag type
	$\Sigma_{(lab)} (lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n)$	(labelled) sum type
	$mod_v.lab$	module projection
	$\lambda var:kind.con$	constructor-level nonrecursive function
	μcon	constructor-level fixpoint
	$con \ con'$	constructor application
	$\{lab_1=con_1, \dots, lab_n=con_n\}$	records of constructors
	$\pi_{lab} con$	record projection
$rdecs ::=$	\cdot	empty
	$rdecs, rdec$	sequence
$rdec ::=$	$lab:con$	record field type
$kind ::=$	Ω	kind of types
	$\{lab_1:kind_1, \dots, lab_n:kind_n\}$	constructor tuple kinds
	$kind \Rightarrow kind'$	constructor function kinds

Figure 1: Constructors and Kinds

The syntax $\langle \dots \rangle$ denotes a phrase which may optionally appear.

2.2 Expressions

$exp ::=$	$scon$	constants
	var	variables
	loc	memory locations
	tag	exception tags
	$fix\ fbnds\ end$	mutually-recursive functions
	$exp\ exp'$	application
	$\{rbnds\}$	record expression
	$\pi_{lab}\ exp$	record projection
	$handle\ exp\ with\ exp'$	handle exception
	$raise^{con}\ exp$	raise exception
	$ref^{con}\ exp$	allocate new ref cell
	$get\ exp$	dereference
	$set\ (exp, exp')$	assignment
	$roll^{con}\ exp$	coerce into μ type
	$unroll\ exp$	coerce from μ type
	$\partial\ exp$	coerce from partial to total function
	$inj_{lab}^{con}\ exp$	injection into sum
	$proj_{lab}^{con}\ exp$	total projection from sum
	$case^{con}\ exp\ of\ exp_1, \dots, exp_n\ end$	sum case analysis
	$new_tag[con]$	extend type Tagged
	$tag(exp, exp')$	injection into type Tagged
	$iftagof\ exp\ is\ exp'\ then\ exp''\ else\ exp'''$	exception tag case analysis
	$mod.lab$	module projection
	$exp_1 =_{Int} exp_2, exp_1 =_{Float} exp_2, \dots$	equalities at base types
$rbnds ::=$	\cdot	empty
	$rbnds, rbnd$	sequence
$rbnd ::=$	$lab = exp$	record field binding
$fbnds ::=$	\cdot	empty
	$fbnds, fbnd$	sequence
$fbnd ::=$	$var'(var:con):con' \mapsto exp$	function binding
$labs ::=$	$lab \mid labs.lab$	sequence of labels
$path ::=$	$var \mid var.labs$	path (qualified variable)

Figure 2: Expressions

2.3 Modules and Signatures

$mod ::=$	var	module variable
	$ [sbnds]$	structure
	$ \lambda var: sig. mod$	functor
	$ mod\ mod'$	functor application
	$ mod. lab$	projection from structure
	$ mod: sig$	signature ascription
$sbnds ::=$	\cdot	structure field bindings
	$ sbnds, sbnd$	
$sbnd ::=$	$lab \triangleright bnd$	
$bnd ::=$	$var = con$	constructor binding
	$ var = exp$	expression binding
	$ var = mod$	module binding
$sig ::=$	$[sdec]$	structure signature
	$ (var: sig) \rightarrow sig'$	partial functor signature
	$ (var: sig) \rightarrow sig'$	total functor signature
$sdec ::=$	\cdot	structure field declarations
	$ sdec, sdec$	
$sdec ::=$	$lab \triangleright dec$	
$dec ::=$	\cdot	declaration lists
	$ decs, dec$	
$dec ::=$	$var: con$	expression variable declaration
	$ var: sig$	module variable declaration
	$ var: kind$	opaque type declaration
	$ var: kind = con$	transparent type declaration
	$ loc: con$	typed locations
	$ tag: con$	typed exception tag

Figure 3: Modules and Signatures

2.4 Useful Abbreviations and Notation

- For readability we often elide the internal names (var 's) when writing out $sbnds$ (and $sdec$ s). In all cases it should be immediately obvious how to consistently restore these with fresh variables.
- We use $sig \rightarrow sig'$ and $sig \rightarrow sig'$ to abbreviate $(var: sig) \rightarrow sig'$ and $(var: sig) \rightarrow sig'$ respectively where var is not free in sig' .
- Many grammatical classes ($labs \Gamma decs \Gamma rdec \Gamma sbnds \Gamma fbn \Gamma sdec \Gamma sbnds \Gamma$ etc.) specify lists of elements. For each of these classes we define a binary append operation written

with a comma. For example we define

$$\begin{aligned} decs, \cdot &::= decs \\ decs, (dec's', dec') &::= (decs, dec's'), dec' \end{aligned}$$

with analogous definitions for all the other classes listed above.

- It is useful to define the following very general syntactic categories of program phrases and program phrase classifiers:

$$\begin{array}{ll} phrase ::= & exp \quad \quad class ::= \quad con \\ & | \quad mod \quad \quad | \quad sig \\ & | \quad con \quad \quad | \quad kno \end{array}$$

- The notation $\{phrase/var\}phrase'$ denotes the capture-free substitution of $phrase$ for free occurrences of var within $phrase'$.
- For consistency we often abbreviate structure bindings $lab \triangleright var = phrase$ as $lab = phrase$ (and similarly signature bindings $lab \triangleright var : class$ as $lab : class$) when the local name is not used. In all cases it should be obvious how to consistently insert fresh variables to correct these omissions.
- We occasionally use the abbreviation $(phrase_i)_{i=1}^n$ as shorthand for $phrase_1, \dots, phrase_n$ (where the phrases are comma-separated).
- The syntactic values of the language are defined in Figure 4. We view each class of syntactic values ($class_v$) as subsets of the corresponding class in the abstract syntax ($class$).

$exp_v ::=$	$scon$	$mod_v ::=$	$path$
	$ loc$		$ [sbnds_v]$
	$ tag$		$ \lambda var: sig.mod$
	$ path$		
	$ \{rbnds_v\}$	$bnd_v ::=$	$var = exp_v$
	$ fix fbnds end$		$ var = mod_v$
	$ \pi_{\bar{k}} fix fbnds end$		$ var = con$
	$ inj_{lab}^{con} exp_v$	$sbnds_v ::=$	\cdot
	$ tag(exp_v, exp_v')$		$ sbnds_v, sbnd_v$
	$ roll^{con} exp_v$		
	$ \partial exp_v$	$sbnd_v ::=$	$lab: bnd_v$
$rbnds_v ::=$	\cdot		
	$ rbnds_v, rbnd_v$		
$rbnd_v ::=$	$lab = exp_v$		
$val ::=$	exp_v		
	$ mod_v$		
	$ con$		

Figure 4: IL values

2.5 Bindings and Scope

We define the functions $BV(\cdot)$ and $dom(\cdot)$ for various bindings Γ declarations Γ and lists thereof:

<i>Function</i>	<i>Definition</i>
$BV(dec)$	$BV(var:con) = var$ $BV(var:knd) = var$ $BV(var:knd=con) = var$ $BV(var:sig) = var$ $BV(loc:con) = loc$ $BV(tag:con) = tag$
$BV(decs)$	$BV(dec_1, \dots, dec_n) = \{BV(dec_1), \dots, BV(dec_n)\}$
$BV(sdecs)$	$BV(lab_1 \triangleright dec_1, \dots, lab_n \triangleright dec_n) = \{BV(dec_1), \dots, BV(dec_n)\}$
$BV(bnd)$	$BV(var=exp) = var$ $BV(var=con) = var$ $BV(var=mod) = var$
$BV(sbnds)$	$BV(lab_1 \triangleright bnd_1, \dots, lab_n \triangleright bnd_n) = \{BV(bnd_1), \dots, BV(bnd_n)\}$
$dom(sdecs)$	$dom(lab_1 \triangleright dec_1, \dots, lab_n \triangleright dec_n) = \{lab_1, \dots, lab_n\}$
$dom(rdecs)$	$dom(lab_1:con_1, \dots, lab_n:con_n) = \{lab_1, \dots, lab_n\}$

The scopes of bound variables are given by the following table:

<i>Binding Phrase</i>	<i>Bound Vars</i>	<i>Scope</i>
$\text{fix } fbnds, var'(var:con):exp \rightarrow, fbnds' \text{ end}$ "	var' var	entire phrase exp
$\lambda var:knd.con$	var	con
$sbnd, sbnds$ $\lambda var:sig.mod$	$BV(sbnd)$ var	$sbnds$ mod
$sdec, sdec$ $var:sig \rightarrow sig'$ $var:sig \rightarrow sig'$	$BV(sdec)$ var var	$sdec$ sig' sig'

We follow standard practice and identify all phrases which differ only with respect to bound variables, locations and exception names. We use the notation $FV(phrase)$ to denote the set of free variables in *phrase*. A phrase is said to be *closed* if it has no free variables (though it may contain free locations or tags).

3 Static Semantics

3.1 Introduction

In this section we define the well-formedness and typing judgments for the internal language. Points of interest include:

- There are no metasyntactic “semantic objects” in the sense of the *Definition*.
- The rules are explicitly formulated so that a judgment holds only if its constituents (declaration lists, etc.) are well-formed.
- The *valuable* expressions always evaluate to a value without side-effects, referencing the store, or raising exceptions. Similarly, valuable modules evaluate without side-effects, referencing the store, or raising exceptions. The purpose of distinguishing total and partial functions, as well as total and partial functors, is to specify which function/functor applications are valuable.

3.2 Notation

- Within the static semantics, optional elements are enclosed by single brackets $\langle \dots \rangle$. Within a single rule, either all such optional elements must occur, or none.

3.3 Judgment Forms

<i>Section</i>	<i>Judgment...</i>	<i>Meaning...</i>
3.4.1	$\vdash \text{decs ok}$ $\text{decs} \vdash \text{dec ok}$	decs is well-formed dec is well-formed
3.4.2	$\text{decs} \vdash \text{bnd} : \text{dec}$	bnd has declaration dec
3.4.3	$\text{decs} \vdash \text{knd} : \text{Kind}$	knd is well-formed
3.4.4	$\text{decs} \vdash \text{con} : \text{knd}$	con has kind knd
3.4.5	$\text{decs} \vdash \text{con} \equiv \text{con}' : \text{knd}$	constructor equivalence at kind knd
3.4.6	$\text{decs} \vdash \text{exp} : \text{con}$	exp has type con
3.4.7	$\text{decs} \vdash \text{sdecs ok}$ $\text{decs} \vdash \text{sig} : \text{Sig}$	sdecs is well-formed sig is well-formed
3.4.8	$\text{decs} \vdash \text{sdecs} \leq \text{sdecs}'$ $\text{decs} \vdash \text{sig} \leq \text{sig}' : \text{Sig}$	component-wise subtyping signature subtyping
3.4.9	$\text{decs} \vdash \text{sdecs} \equiv \text{sdecs}'$ $\text{decs} \vdash \text{sig} \equiv \text{sig}' : \text{Sig}$	component-wise equivalence signature equivalence
3.4.10	$\text{decs} \vdash \text{sbnds} : \text{sdecs}$ $\text{decs} \vdash \text{mod} : \text{sig}$	sbnds has declaration list sdecs mod has signature sig
3.4.11	$\text{decs} \vdash \text{exp} \downarrow \text{con}$ $\text{decs} \vdash \text{mod} \downarrow \text{sig}$ $\text{decs} \vdash \text{exp} \downarrow$ $\text{decs} \vdash \text{sbnds} \downarrow$ $\text{decs} \vdash \text{mod} \downarrow$	exp is valuable with type con mod is valuable with signature sig exp is valuable sbnds is valuable mod is valuable

3.4 Inference Rules

3.4.1 Well-formed Declarations

$$\boxed{\vdash \text{decs ok}}$$

$$\frac{}{\vdash \cdot \text{ok}} \quad (1)$$

$$\frac{\text{decs} \vdash \text{dec ok}}{\vdash \text{decs}, \text{dec ok}} \quad (2)$$

$$\boxed{\text{decs} \vdash \text{dec ok}}$$

$$\frac{\text{decs} \vdash \text{knd} : \text{Kind} \quad \text{var} \notin \text{BV}(\text{decs})}{\text{decs} \vdash \text{var} : \text{knd ok}} \quad (3)$$

$$\frac{\text{decs} \vdash \text{con} : \text{knd} \quad \text{var} \notin \text{BV}(\text{decs})}{\text{decs} \vdash \text{var} : \text{knd} = \text{con ok}} \quad (4)$$

$$\frac{\text{decs} \vdash \text{con} : \Omega \quad \text{var} \notin \text{BV}(\text{decs})}{\text{decs} \vdash \text{var} : \text{con ok}} \quad (5)$$

$$\frac{\text{decs} \vdash \text{sig} : \text{Sig} \quad \text{var} \notin \text{BV}(\text{decs})}{\text{decs} \vdash \text{var} : \text{sig ok}} \quad (6)$$

$$\frac{\text{decs} \vdash \text{con} \equiv \text{con}' \text{ Ref} : \Omega \quad \text{loc} \notin \text{BV}(\text{decs})}{\text{decs} \vdash \text{loc} : \text{con ok}} \quad (7)$$

$$\frac{\text{decs} \vdash \text{con} \equiv \text{con}' \text{ Tag} : \Omega \quad \text{tag} \notin \text{BV}(\text{decs})}{\text{decs} \vdash \text{tag} : \text{con ok}} \quad (8)$$

3.4.2 Well-formed Bindings

$$\boxed{\text{decs} \vdash \text{bnd} : \text{dec}}$$

$$\frac{\text{decs} \vdash \text{con} : \text{knd}}{\text{decs} \vdash \text{var} = \text{con} : \text{var} : \text{knd}} \quad (9)$$

$$\frac{\text{decs} \vdash \text{con} \equiv \text{con}' : \text{knd}}{\text{decs} \vdash \text{var} = \text{con} : \text{var} : \text{knd} = \text{con}'} \quad (10)$$

$$\frac{\text{decs} \vdash \text{exp} : \text{con}}{\text{decs} \vdash \text{var} = \text{exp} : \text{var} : \text{con}} \quad (11)$$

$$\frac{\text{decs} \vdash \text{mod} : \text{sig}' \quad \text{decs} \vdash \text{sig}' \leq \text{sig} : \text{Sig}}{\text{decs} \vdash \text{var} = \text{mod} : \text{var} : \text{sig}} \quad (12)$$

3.4.3 Well-formed Kinds

$$\boxed{decs \vdash kind : Kind}$$

$$\frac{}{decs \vdash \Omega : Kind} \quad (13)$$

$$\frac{lab_1, \dots, lab_n \text{ distinct} \quad n \geq 0}{decs \vdash \{lab_1:kind_1, \dots, lab_n:kind_n\} : Kind} \quad (14)$$

$$\frac{decs \vdash kind : Kind \quad decs \vdash kind' : Kind}{decs \vdash kind \Rightarrow kind' : Kind} \quad (15)$$

3.4.4 Well-formed Constructors

$$\boxed{decs \vdash con : kind}$$

$$\frac{\vdash decs \text{ ok} \quad decs = decs', var:kind \langle =con \rangle, decs''}{decs \vdash var : kind} \quad (16)$$

$$\frac{}{decs \vdash Tagged : \Omega} \quad (17)$$

$$\frac{decs \vdash con : \Omega}{decs \vdash con \text{ Ref} : \Omega} \quad (18)$$

$$\frac{decs \vdash con : \Omega}{decs \vdash con \text{ Tag} : \Omega} \quad (19)$$

$$\frac{decs \vdash con : \Omega \quad decs \vdash con' : \Omega}{decs \vdash con \rightarrow con' : \Omega} \quad (20)$$

$$\frac{decs \vdash con : \Omega \quad decs \vdash con' : \Omega}{decs \vdash con \rightarrow con' : \Omega} \quad (21)$$

$$\frac{lab_1, \dots, lab_n \text{ distinct} \quad decs \vdash con_1 : \Omega \quad \dots \quad decs \vdash con_n : \Omega}{decs \vdash \{lab_1:con_1, \dots, lab_n:con_n\} : \Omega} \quad (22)$$

$$\frac{\langle i \in 1..n \rangle \quad lab_1, \dots, lab_n \text{ distinct} \quad decs \vdash con_1 : \Omega \quad \dots \quad decs \vdash con_n : \Omega}{decs \vdash \Sigma_{\langle lab_i \rangle} (lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n) : \Omega} \quad (23)$$

$$\frac{lab_1, \dots, lab_n \text{ distinct} \quad n \geq 0 \quad decs \vdash con_1 : kind_1 \quad \dots \quad decs \vdash con_n : kind_n}{decs \vdash \{lab_1=con_1, \dots, lab_n=con_n\} : \{lab_1:kind_1, \dots, lab_n:kind_n\}} \quad (24)$$

$$\frac{i \in 1..n \quad decs \vdash con : \{lab_1:knd_1, \dots, lab_n:knd_n\}}{decs \vdash \pi_{lab_i} con : knd_i} \quad (25)$$

$$\frac{decs, var:knd \vdash con : knd'}{decs \vdash \lambda var:knd. con : knd \Rightarrow knd'} \quad (26)$$

$$\frac{decs \vdash con : knd \Rightarrow knd}{decs \vdash \mu con : knd} \quad (27)$$

$$\frac{decs \vdash con : knd' \Rightarrow knd \quad decs \vdash con' : knd'}{decs \vdash con con' : knd} \quad (28)$$

$$\frac{decs \vdash mod_v : [sdecs, lab \triangleright var:knd, sdecs']}{decs \vdash mod_v.lab : knd} \quad (29)$$

3.4.5 Constructor Equivalence

$$\boxed{decs \vdash con \equiv con' : knd}$$

$$\frac{\vdash decs \text{ ok} \quad decs = decs', var:knd=con, decs''}{decs \vdash var \equiv con : knd} \quad (30)$$

Rule 30: The well-formedness judgment $\vdash decs \text{ ok}$ guarantees that $FV(con) \cap BV(decs'') = \emptyset$.

$$\frac{decs \vdash mod_v : [sdecs, lab:knd=con, sdecs'] \quad BV(sdecs) \cap FV(con) = \emptyset}{decs \vdash mod_v.lab \equiv con : knd} \quad (31)$$

Rule 31: The projection must yield a valid constructor with respect to the ambient context, $decs$. This can always be arranged through use of the self rules (Rules 101 and 102).

$$\frac{decs \vdash con_1 \equiv con_2 : \Omega \quad decs \vdash con'_1 \equiv con'_2 : \Omega}{decs \vdash con_1 \multimap con'_1 \equiv con_2 \multimap con'_2 : \Omega} \quad (32)$$

$$\frac{decs \vdash con_1 \equiv con_2 : \Omega \quad decs \vdash con'_1 \equiv con'_2 : \Omega}{decs \vdash con_1 \rightarrow con'_1 \equiv con_2 \rightarrow con'_2 : \Omega} \quad (33)$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash con \text{ Ref} \equiv con' \text{ Ref} : \Omega} \quad (34)$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash con \text{ Tag} \equiv con' \text{ Tag} : \Omega} \quad (35)$$

$$\frac{lab_1, \dots, lab_n \text{ distinct} \quad \forall i \in 1..n : decs \vdash con_i \equiv con'_i : \Omega}{decs \vdash \{lab_1:con_1, \dots, lab_n:con_n\} \equiv \{lab_1:con'_1, \dots, lab_n:con'_n\} : \Omega} \quad (36)$$

Rule 36: To be equivalent, two IL record types must have equivalent components with the same labels in the same order.

$$\frac{i \in 1..n \quad decs \vdash con \equiv con' : \{lab_1:knd_1, \dots, lab_n:knd_n\}}{decs \vdash \pi_{lab_i} con \equiv \pi_{lab_i} con' : knd_i} \quad (37)$$

$$\frac{i \in 1..n \quad decs \vdash con \equiv \{lab_1:con_1, \dots, lab_n:con_n\} : \{lab_1:knd_1, \dots, lab_n:knd_n\}}{decs \vdash \pi_{lab_i} con \equiv con_i : knd_i} \quad (38)$$

$$\frac{decs \vdash con \equiv con' : knd \Rightarrow knd}{decs \vdash \mu con \equiv \mu con' : knd} \quad (39)$$

$$\frac{\langle i \in 1..n \rangle \quad decs \vdash con_1 \equiv con'_1 : \Omega \quad \dots \quad decs \vdash con_n \equiv con'_n : \Omega}{decs \vdash \Sigma_{\langle lab_i \rangle} (lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n) \equiv \Sigma_{\langle lab_i \rangle} (lab_1 \mapsto con'_1, \dots, lab_n \mapsto con'_n) : \Omega} \quad (40)$$

$$\frac{decs \vdash con_1 \equiv con_2 : knd' \Rightarrow knd \quad decs \vdash con'_1 \equiv con'_2 : knd'}{decs \vdash con_1 con_2 \equiv con'_1 con'_2 : knd} \quad (41)$$

$$\frac{decs, var:knd' \vdash con : knd \quad decs \vdash con : knd'}{decs \vdash (\lambda var:knd'. con) con' \equiv \{con'/var\} con : knd} \quad (42)$$

$$\frac{decs \vdash con : knd}{decs \vdash con \equiv con : knd} \quad (43)$$

$$\frac{decs \vdash con' \equiv con : knd}{decs \vdash con \equiv con' : knd} \quad (44)$$

$$\frac{decs \vdash con \equiv con' : knd \quad decs \vdash con' \equiv con'' : knd}{decs \vdash con \equiv con'' : knd} \quad (45)$$

3.4.6 Well-formed Expressions

$decs \vdash exp : con$

$$\frac{\vdash decs \text{ ok} \quad decs = decs', var:con, decs''}{decs \vdash var : con} \quad (46)$$

$$\frac{\vdash decs \text{ ok} \quad decs = decs', loc:con, decs''}{decs \vdash loc : con} \quad (47)$$

$$\frac{\begin{array}{c} \vdash \text{decs ok} \\ \text{decs} = \text{decs}', \text{tag} : \text{con}, \text{decs}'' \end{array}}{\text{decs} \vdash \text{tag} : \text{con}} \quad (48)$$

$$\frac{\text{decs} \vdash \text{exp} : \text{con}' \rightarrow \text{con} \quad \text{decs} \vdash \text{exp}' : \text{con}'}{\text{decs} \vdash \text{exp exp}' : \text{con}} \quad (49)$$

$$\frac{\text{decs} \vdash \text{exp} : \text{con}' \rightarrow \text{con} \quad \text{decs} \vdash \text{exp}' : \text{con}'}{\text{decs} \vdash \text{exp exp}' : \text{con}} \quad (50)$$

$$\frac{\begin{array}{c} \forall i \in 1..n : \text{decs}, (\text{var}'_i : \text{con}_i \rightarrow \text{con}'_i)_{i=1}^n, \text{var}_i : \text{con}_i \vdash \text{exp}_i : \text{con}'_i \\ \text{Rule 52 does not apply.} \end{array}}{\text{decs} \vdash \text{fix} (\text{var}'_i (\text{var}_i : \text{con}_i) : \text{con}'_i \mapsto \text{exp}_i)_{i=1}^n \text{ end} : \{1 : \text{con}_1 \rightarrow \text{con}'_1, \dots, n : \text{con}_n \rightarrow \text{con}'_n\}} \quad (51)$$

$$\frac{\begin{array}{c} \text{var}' \notin \text{FV exp} \\ \text{decs}, \text{var} : \text{con} \vdash \text{exp} \downarrow \text{con}' \end{array}}{\text{decs} \vdash \text{fix var}' (\text{var} : \text{con}) : \text{con}' \mapsto \text{exp end} : \{1 : \text{con} \rightarrow \text{con}'\}} \quad (52)$$

$$\frac{\begin{array}{c} \text{lab}_1, \dots, \text{lab}_n \text{ distinct} \\ \text{decs} \vdash \text{exp}_1 : \text{con}_1 \quad \dots \quad \text{decs} \vdash \text{exp}_n : \text{con}_n \end{array}}{\text{decs} \vdash \{\text{lab}_1 = \text{exp}_1, \dots, \text{lab}_n = \text{exp}_n\} : \{\text{lab}_1 : \text{con}_1, \dots, \text{lab}_n : \text{con}_n\}} \quad (53)$$

Rule 53: For the record bindings to be well-typed with respect to the declarations, they must have the same labels in the same order (with no duplicate labels) and components must be well-typed elementwise.

$$\frac{\text{decs} \vdash \text{exp} : \{\text{rdecs}, \text{lab} : \text{con}, \text{rdecs}'\}}{\text{decs} \vdash \pi_{\text{lab}} \text{exp} : \text{con}} \quad (54)$$

$$\frac{\text{decs} \vdash \text{exp} : \text{con} \quad \text{decs} \vdash \text{exp}' : \text{Tagged} \rightarrow \text{con}}{\text{decs} \vdash \text{handle exp with exp}' : \text{con}} \quad (55)$$

$$\frac{\text{decs} \vdash \text{exp} : \text{Tagged} \quad \text{decs} \vdash \text{con} : \Omega}{\text{decs} \vdash \text{raise}^{\text{con}} \text{exp} : \text{con}} \quad (56)$$

$$\frac{\text{decs} \vdash \text{con} : \Omega}{\text{decs} \vdash \text{new_tag}[\text{con}] : \text{con Tag}} \quad (57)$$

$$\frac{\text{decs} \vdash \text{exp} : \text{con}}{\text{decs} \vdash \text{ref}^{\text{con}} \text{exp} : \text{con Ref}} \quad (58)$$

$$\frac{\text{decs} \vdash \text{exp} : \text{con Ref}}{\text{decs} \vdash \text{get exp} : \text{con}} \quad (59)$$

$$\frac{\begin{array}{c} \text{decs} \vdash \text{exp} : \text{con Ref} \\ \text{decs} \vdash \text{exp}' : \text{con} \end{array}}{\text{decs} \vdash \text{set} (\text{exp}, \text{exp}') : \text{Unit}} \quad (60)$$

$$\frac{\begin{array}{l} decs \vdash con \equiv (\pi_{lab} (\mu con')) \langle con'' \rangle : \Omega \\ decs \vdash exp : (\pi_{lab} (con' (\mu con'))) \langle con'' \rangle \end{array}}{decs \vdash roll^{con} exp : con} \quad (61)$$

$$\frac{\begin{array}{l} decs \vdash con \equiv (\pi_{lab} (\mu con')) \langle con'' \rangle : \Omega \\ decs \vdash exp : con \end{array}}{decs \vdash unroll exp : (\pi_{lab} (con' (\mu con'))) \langle con'' \rangle} \quad (62)$$

$$\frac{decs \vdash exp : con \rightarrow con'}{decs \vdash \partial exp : con \rightarrow con'} \quad (63)$$

$$\frac{\begin{array}{l} i \in 1..n \\ con = \Sigma_{lab_i} (lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n) \\ decs \vdash exp : con_i \end{array}}{decs \vdash inj_{lab_i}^{con} exp : con} \quad (64)$$

$$\frac{\begin{array}{l} i \in 1..n \\ decs \vdash exp : \Sigma_{lab_i} (lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n) \end{array}}{decs \vdash proj_{lab_i}^{\Sigma_{lab_i} (lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n)} exp : con_i} \quad (65)$$

$$\frac{\begin{array}{l} n \geq 1 \\ con = \Sigma (lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n) \\ decs \vdash exp : con \\ \forall i \in 1..n : decs \vdash exp_i : \Sigma_{lab_i} (lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n) \rightarrow con' \end{array}}{decs \vdash case^{con} exp \text{ of } exp_1, \dots, exp_n \text{ end} : con'} \quad (66)$$

$$\frac{decs \vdash exp : con \text{ Tag} \quad decs \vdash exp' : con}{decs \vdash tag(exp, exp') : Tagged} \quad (67)$$

$$\frac{\begin{array}{l} decs \vdash exp : Tagged \quad decs \vdash exp' : con \text{ Tag} \\ decs \vdash exp'' : con \rightarrow con' \quad decs \vdash exp''' : con' \end{array}}{decs \vdash iftagof exp \text{ is } exp' \text{ then } exp'' \text{ else } exp''' : con'} \quad (68)$$

$$\frac{\begin{array}{l} decs \vdash mod : [sdecs, lab:con, sdecs'] \\ BV(sdecs) \cap FV(con) = \emptyset \end{array}}{decs \vdash mod.lab : con} \quad (69)$$

Rule 69: A projection is only well-formed if the result is typable. If mod is a value, the projection is always well-formed.

$$\frac{decs \vdash exp : con' \quad decs \vdash con \equiv con' : \Omega}{decs \vdash exp : con} \quad (70)$$

3.4.7 Well-formed Signatures

$$\boxed{decs \vdash sdecs \text{ ok}}$$

$$\frac{\vdash decs \text{ ok}}{decs \vdash \cdot \text{ ok}} \quad (71)$$

$$\frac{\begin{array}{l} decs \vdash dec \text{ ok} \\ decs, dec \vdash sdecs \text{ ok} \\ lab \notin \text{dom}(sdecs) \end{array}}{decs \vdash lab \triangleright dec, sdecs \text{ ok}} \quad (72)$$

$$\boxed{decs \vdash sig : \text{Sig}}$$

$$\frac{decs \vdash sdecs \text{ ok}}{decs \vdash [sdecs] : \text{Sig}} \quad (73)$$

$$\frac{decs, var: sig \vdash sig' : \text{Sig}}{decs \vdash var: sig \rightarrow sig' : \text{Sig}} \quad (74)$$

$$\frac{decs, var: sig \vdash sig' : \text{Sig}}{decs \vdash var: sig \rightarrow sig' : \text{Sig}} \quad (75)$$

3.4.8 Signature Subtyping

$$\boxed{decs \vdash sdecs \leq sdecs'}$$

$$\overline{decs \vdash \cdot \leq \cdot} \quad (76)$$

$$\frac{decs, var: knd = con \vdash sdecs \leq sdecs'}{decs \vdash lab \triangleright var: knd = con, sdecs \leq lab \triangleright var: knd, sdecs'} \quad (77)$$

$$\frac{\begin{array}{l} decs \vdash sig \leq sig' : \text{Sig} \\ decs, var: sig \vdash sdecs \leq sdecs' \end{array}}{decs \vdash lab \triangleright var: sig, sdecs \leq lab \triangleright var: sig', sdecs'} \quad (78)$$

$$\frac{\begin{array}{l} decs \vdash lab: dec \equiv lab: dec' \\ decs, dec \vdash sdecs \leq sdecs' \end{array}}{decs \vdash lab \triangleright dec, sdecs \leq lab \triangleright dec', sdecs'} \quad (79)$$

$$\boxed{decs \vdash sig \leq sig' : \text{Sig}}$$

$$\frac{decs \vdash sdecs \leq sdecs'}{decs \vdash [sdecs] \leq [sdecs'] : \text{Sig}} \quad (80)$$

$$\frac{decs \vdash sig_2 \leq sig_1 : \text{Sig} \quad decs, var: sig_2 \vdash sig'_1 \leq sig'_2 : \text{Sig}}{decs \vdash var: sig_1 \rightarrow sig'_1 \leq var: sig_2 \rightarrow sig'_2 : \text{Sig}} \quad (81)$$

$$\frac{decs \vdash sig_2 \leq sig_1 : \text{Sig} \quad decs, var: sig_2 \vdash sig'_1 \leq sig'_2 : \text{Sig}}{decs \vdash var: sig_1 \rightarrow sig'_1 \leq var: sig_2 \rightarrow sig'_2 : \text{Sig}} \quad (82)$$

$$\frac{decs \vdash sig_2 \leq sig_1 : \text{Sig} \quad decs, var: sig_2 \vdash sig'_1 \leq sig'_2 : \text{Sig}}{decs \vdash var: sig_1 \rightarrow sig'_1 \leq var: sig_2 \rightarrow sig'_2 : \text{Sig}} \quad (83)$$

3.4.9 Signature Equivalence

$$\boxed{decs \vdash sdecs \equiv sdecs'}$$

$$\frac{}{decs \vdash \cdot \equiv \cdot} \quad (84)$$

$$\frac{decs, var: kind \vdash sdecs \equiv sdecs'}{decs \vdash lab \triangleright var: kind, sdecs \equiv lab \triangleright var: kind, sdecs'} \quad (85)$$

$$\frac{decs \vdash con \equiv con' : kind \quad decs, var: kind = con \vdash sdecs \equiv sdecs'}{decs \vdash lab \triangleright var: kind = con, sdecs \equiv lab \triangleright var: kind = con', sdecs'} \quad (86)$$

$$\frac{decs \vdash con \equiv con' : \Omega \quad decs, var: con \vdash sdecs \equiv sdecs'}{decs \vdash lab \triangleright var: con, sdecs \equiv lab \triangleright var: con', sdecs'} \quad (87)$$

$$\frac{decs \vdash sig \equiv sig' : kind \quad decs, var: sig \vdash sdecs \equiv sdecs'}{decs \vdash lab \triangleright var: sig, sdecs \equiv lab \triangleright var: sig', sdecs'} \quad (88)$$

$$\boxed{decs \vdash sig \equiv sig' : \text{Sig}}$$

$$\frac{decs \vdash sdecs \equiv sdecs'}{decs \vdash [sdecs] \equiv [sdecs'] : \text{Sig}} \quad (89)$$

$$\frac{decs \vdash sig_1 \equiv sig_2 : \text{Sig} \quad decs, var: sig_1 \vdash sig'_1 \equiv sig'_2 : \text{Sig}}{decs \vdash var: sig_1 \rightarrow sig'_1 \equiv var: sig_2 \rightarrow sig'_2 : \text{Sig}} \quad (90)$$

$$\frac{decs \vdash sig_1 \equiv sig_2 : \text{Sig} \quad decs, var: sig_1 \vdash sig'_1 \equiv sig'_2 : \text{Sig}}{decs \vdash var: sig_1 \rightarrow sig'_1 \equiv var: sig_2 \rightarrow sig'_2 : \text{Sig}} \quad (91)$$

3.4.10 Well-formed Modules

$$\boxed{decs \vdash sbnds : sdecs}$$

$$\frac{}{decs \vdash \cdot : \cdot} \quad (92)$$

$$\frac{decs \vdash bnd : dec \quad decs, dec \vdash sbnds : sdecs}{decs \vdash lab \triangleright bnd, sbnds : lab \triangleright dec, sdecs} \quad (93)$$

$$\boxed{decs \vdash mod : sig}$$

$$\frac{\vdash decs \text{ ok} \quad decs = decs', var : sig, decs''}{decs \vdash var : sig} \quad (94)$$

$$\frac{decs \vdash sbnds : sdecs}{decs \vdash [sbnds] : [sdecs]} \quad (95)$$

$$\frac{decs, var : sig \vdash mod : sig'}{decs \vdash \lambda var : sig. mod : var : sig \rightarrow sig'} \quad (96)$$

$$\frac{decs, var : sig \vdash mod \downarrow sig'}{decs \vdash \lambda var : sig. mod : var : sig \rightarrow sig'} \quad (97)$$

$$\frac{decs \vdash mod : sig' \rightarrow sig \quad decs \vdash mod' : sig'}{decs \vdash mod \ mod' : sig} \quad (98)$$

Rule 98: Only functors with non-dependent types may be applied. Dependencies can be eliminated by uses of the subtyping and equivalence rules. If the argument is a value, dependencies can always be eliminated.

$$\frac{decs \vdash mod : [sdecs, lab : sig, sdecs'] \quad BV(sdecs) \cap FV(sig) = \emptyset}{decs \vdash mod.lab : sig} \quad (99)$$

Rule 99: A projection is only well-formed if the result can be given a signature in the ambient context. If mod is a value, this can always occur.

$$\frac{decs \vdash mod : sig}{decs \vdash mod : sig : sig} \quad (100)$$

Rule 100: Ascription of a signature to a module can make types in mod abstract by forgetting equations.

$$\frac{decs \vdash mod_v : [sdecs, lab \triangleright var : kind, sdecs']}{decs \vdash mod_v : [sdecs, lab \triangleright var : kind = mod_v.lab, sdecs']} \quad (101)$$

Rule 101: The “self” rule. If $mod.labs$ specifies a type and mod has a well-defined value then $mod.labs \equiv mod.labs$; we add this fact to the signature.

$$\frac{\begin{array}{c} decs \vdash mod_v : [sdecs, lab \triangleright var: sig, sdecs'] \\ decs \vdash mod_v.lab : sig' \end{array}}{decs \vdash mod_v : [sdecs, lab \triangleright var: sig', sdecs']} \quad (102)$$

Rule 102: The “self” rule can be recursively applied.

$$\frac{decs \vdash mod : sig \quad decs \vdash sig \leq sig' : \mathbf{Sig}}{decs \vdash mod : sig'} \quad (103)$$

3.4.11 Valuability

$$\frac{decs \vdash exp : con \quad decs \vdash exp \downarrow}{decs \vdash exp \downarrow con} \quad (104)$$

$$\frac{decs \vdash mod : sig \quad decs \vdash mod \downarrow}{decs \vdash mod \downarrow sig} \quad (105)$$

$$\overline{decs \vdash exp_v \downarrow} \quad (106)$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lab \downarrow} \quad (107)$$

$$\frac{decs \vdash exp_1 \downarrow con' \rightarrow con \quad decs \vdash exp_2 \downarrow}{decs \vdash exp_1 exp_2 \downarrow} \quad (108)$$

$$\frac{decs \vdash exp_1 \downarrow \quad \dots \quad decs \vdash exp_n \downarrow}{decs \vdash \{lab_1 = exp_1, \dots, lab_n = exp_n\} \downarrow} \quad (109)$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash \pi_{lab} exp \downarrow} \quad (110)$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash roll^{con} exp \downarrow} \quad (111)$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash unroll exp \downarrow} \quad (112)$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash \text{inj}_{lab}^{con} exp \downarrow} \quad (113)$$

$$\frac{}{decs \vdash \text{proj}_{lab}^{con} exp \downarrow} \quad (114)$$

$$\frac{decs \vdash exp \downarrow \quad decs \vdash exp' \downarrow}{decs \vdash \text{tag}(exp, exp') \downarrow} \quad (115)$$

$$\frac{decs \vdash exp \downarrow \quad decs \vdash exp_1 \downarrow \quad \dots \quad decs \vdash exp_n \downarrow}{decs \vdash \text{case}^{con} exp \text{ of } exp_1, \dots, exp_n \text{ end} \downarrow} \quad (116)$$

$$\boxed{decs \vdash sbnds \downarrow} \quad (117)$$

$$\frac{decs \vdash exp \downarrow con \quad decs, var:con \vdash sbnds \downarrow}{decs \vdash lab \triangleright var = exp, sbnds \downarrow} \quad (118)$$

$$\frac{decs \vdash con : kn d \quad decs, var:kn d = con \vdash sbnds \downarrow}{decs \vdash lab \triangleright var = con, sbnds \downarrow} \quad (119)$$

$$\frac{decs \vdash mod \downarrow sig \quad decs, var:sig \vdash sbnds \downarrow}{decs \vdash lab \triangleright var = mod, sbnds \downarrow} \quad (120)$$

$$\boxed{decs \vdash mod \downarrow} \quad (121)$$

$$\frac{}{decs \vdash mod_v \downarrow} \quad (122)$$

$$\frac{decs \vdash sbnds \downarrow}{decs \vdash [sbnds] \downarrow} \quad (122)$$

$$\frac{decs \vdash mod \downarrow sig' \rightarrow sig \quad decs \vdash mod' \downarrow}{decs \vdash mod \ mod' \downarrow} \quad (123)$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lab \downarrow} \quad (124)$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod:sig \downarrow} \quad (125)$$

4 Dynamic Semantics

4.1 Introduction

The dynamic semantics of the internal language is a call-by-value operational semantics presented as a rewriting system on states of an abstract machine. The presentation is strongly influenced by the work of Plotkin [Plo81] and Wright and Felleisen [WF91] and is a significant departure from the framework employed in *The Definition*. In particular we employ a *small step* semantics in which transitions represent basic evaluation steps of an abstract machine. We rely on substitution rather than environments; values are particular expressions of the language. Exception propagation is handled by explicitly maintaining the evaluation context rather than relying on implicit rules for exception propagation. We maintain a store for assignable cells as in *The Definition* and a typing context which types locations and dynamically-created tags associated with values of type `Tagged`; these are the only extra-linguistic constructs in the formalism. As Wright and Felleisen have demonstrated these could be made part of the language by introducing a `letref` construct.

Each state Σ of the abstract machine is a four-tuple of the form

$$(\Delta, \sigma, E, phrase),$$

where

- Δ is a typing context (*decs*) for locations and tags created at run-time. This maintains a record of what exception tags and locations have been allocated and is also used in our soundness proofs.
- σ is a finite mapping from locations (*loc's*) typed in Δ to expression values (*exp_v*). The syntax for all internal language values appeared in Figure 4.
- E is a stack of evaluation frames (see Figure 5). This represents an implementation's control stack or equivalently the current continuation. Appending two stacks corresponds to composition of continuations; accordingly we write $E \circ E'$ for the concatenation of E and E' defined in the obvious way. The meta-variable R ranges over control stacks that do not contain a frame of the form "handle [] with *exp*".
- *phrase* is an expression, module or constructor.

We let Σ_t range over the set of terminal states where a state is called *terminal* if it has one of the forms:

$$\begin{array}{ll} (\Delta, \sigma, [], val) & \text{normal termination} \\ (\Delta, \sigma, [], \text{raise}^{con} exp_v) & \text{uncaught exception} \end{array}$$

All other states are called *nonterminal*.

The dynamic semantics is a transition relation $\Sigma \hookrightarrow \Sigma'$ between states. As usual we denote the reflexive-transitive closure of \hookrightarrow by \hookrightarrow^* .

$$\begin{aligned}
F ::= & \quad [] \exp \\
& \quad | \exp_v [] \\
& \quad | \{rbnds_v, lab=[], rbnds\} \\
& \quad | \pi_{lab} [] \\
& \quad | \text{handle } [] \text{ with } \exp \\
& \quad | \text{raise}^{con} [] \\
& \quad | \text{ref}^{con} [] \\
& \quad | \text{get } [] \\
& \quad | \text{set } ([], \exp) \\
& \quad | \text{set } (\exp_v, []) \\
& \quad | \text{roll}^{con} [] \\
& \quad | \text{unroll } [] \\
& \quad | \partial [] \\
& \quad | \text{inj}_i^{con} [] \\
& \quad | \text{proj}_i^{con} [] \\
& \quad | \text{case}^{con} [] \text{ of } \exp_1, \dots, \exp_n \text{ end} \\
& \quad | \text{tag}(\exp_v, []) \\
& \quad | \text{iftagof } [] \text{ is } \exp' \text{ then } \exp'' \text{ else } \exp''' \\
& \quad | \text{iftagof } \exp_v \text{ is } [] \text{ then } \exp'' \text{ else } \exp''' \\
& \quad | [] . lab \\
& \quad | [sbnds_v, lab \triangleright var=[], sbnds] \\
& \quad | [] \text{ mod} \\
& \quad | \text{mod}_v [] \\
& \quad | [] : sig \\
\\
E ::= & \quad [] \\
& \quad | E \circ F
\end{aligned}$$

Figure 5: Evaluation Frames

4.2 Transition Relation

4.2.1 Search for Next Redex

$$(\Delta, \sigma, E, \exp \exp') \hookrightarrow (\Delta, \sigma, E \circ ([\exp']), \exp) \quad (126)$$

$$(\Delta, \sigma, E \circ ([\exp]), \exp_v) \hookrightarrow (\Delta, \sigma, E \circ (\exp_v []), \exp) \quad (127)$$

$$\begin{aligned}
& (\Delta, \sigma, E, \{rbnds_v, lab=\exp, rbnds\}) \hookrightarrow \\
& (\Delta, \sigma, E \circ \{rbnds_v, lab=[], rbnds\}, \exp) \quad (128) \\
& \quad \text{if } \exp \text{ is not a value}
\end{aligned}$$

$$(\Delta, \sigma, E \circ \{rbnds_v, lab=[], rbnds_v'\}, \exp_v) \hookrightarrow (\Delta, \sigma, E, \{rbnds_v, lab=\exp_v, rbnds_v'\}) \quad (129)$$

$$\begin{aligned}
(\Delta, \sigma, E \circ \{rbnds_v, lab=[], rbnds_v', lab'=exp', rbnds\}, exp_v) &\hookrightarrow \\
(\Delta, \sigma, E \circ \{rbnds_v, lab=exp_v, rbnds_v', lab'=[], rbnds\}, exp') &\hookrightarrow \\
&\text{if } exp' \text{ is not a value}
\end{aligned} \tag{130}$$

$$\begin{aligned}
(\Delta, \sigma, E, \pi_{lab} exp) &\hookrightarrow (\Delta, \sigma, E \circ (\pi_{lab} []), exp) \\
&\text{if } exp' \text{ is not a fix value}
\end{aligned} \tag{131}$$

$$(\Delta, \sigma, E, \text{handle } exp \text{ with } exp') \hookrightarrow (\Delta, \sigma, E \circ (\text{handle } [] \text{ with } exp'), exp) \tag{132}$$

$$\begin{aligned}
(\Delta, \sigma, E, \text{raise}^{con} exp) &\hookrightarrow (\Delta, \sigma, E \circ (\text{raise}^{con} []), exp) \\
&\text{if } exp \text{ is not a value}
\end{aligned} \tag{133}$$

$$(\Delta, \sigma, E \circ (\text{raise}^{con} []), exp_v) \hookrightarrow (\Delta, \sigma, E, \text{raise}^{con} exp_v) \tag{134}$$

$$(\Delta, \sigma, E, \text{ref}^{con} exp) \hookrightarrow (\Delta, \sigma, E \circ (\text{ref}^{con} []), exp) \tag{135}$$

$$(\Delta, \sigma, E, \text{get } exp) \hookrightarrow (\Delta, \sigma, E \circ (\text{get} []), exp) \tag{136}$$

$$(\Delta, \sigma, E, \text{set}(exp, exp')) \hookrightarrow (\Delta, \sigma, E \circ (\text{set}([], exp')), exp) \tag{137}$$

$$(\Delta, \sigma, E \circ (\text{set}([], exp)), exp_v) \hookrightarrow (\Delta, \sigma, E \circ (\text{set}(exp_v, [])), exp) \tag{138}$$

$$\begin{aligned}
(\Delta, \sigma, E, \text{roll}^{con} exp) &\hookrightarrow (\Delta, \sigma, E \circ (\text{roll}^{con} []), exp) \\
&\text{if } exp \text{ is not a value}
\end{aligned} \tag{139}$$

$$(\Delta, \sigma, E \circ (\text{roll}^{con} []), exp_v) \hookrightarrow (\Delta, \sigma, E, \text{roll}^{con} exp_v) \tag{140}$$

$$(\Delta, \sigma, E, \text{unroll } exp) \hookrightarrow (\Delta, \sigma, E \circ (\text{unroll} []), exp) \tag{141}$$

$$\begin{aligned}
(\Delta, \sigma, E, \partial exp) &\hookrightarrow (\Delta, \sigma, E \circ (\partial []), exp) \\
&\text{if } exp \text{ is not a value}
\end{aligned} \tag{142}$$

$$(\Delta, \sigma, E \circ (\partial []), exp_v) \hookrightarrow (\Delta, \sigma, E, \partial exp_v) \tag{143}$$

$$\begin{aligned}
(\Delta, \sigma, E, \text{inj}_{lab}^{con} exp) &\hookrightarrow (\Delta, \sigma, E \circ (\text{inj}_{lab}^{con} []), exp) \\
&\text{if } exp \text{ is not a value}
\end{aligned} \tag{144}$$

$$(\Delta, \sigma, E \circ (\text{inj}_{lab}^{con} []), exp_v) \hookrightarrow (\Delta, \sigma, E, \text{inj}_{lab}^{con} exp_v) \tag{145}$$

$$(\Delta, \sigma, E, \text{proj}_{lab}^{con} exp) \hookrightarrow (\Delta, \sigma, E \circ (\text{proj}_{lab}^{con} []), exp) \tag{146}$$

$$(\Delta, \sigma, E, \text{tag}(exp, exp')) \hookrightarrow (\Delta, \sigma, E \circ (\text{tag}([], exp')), exp) \quad \text{if } exp \text{ or } exp' \text{ is not a value} \quad (147)$$

$$(\Delta, \sigma, E \circ \text{tag}([], exp), exp_v) \hookrightarrow (\Delta, \sigma, E \circ (\text{tag}(exp_v, [])), exp) \quad (148)$$

$$(\Delta, \sigma, E \circ (\text{tag}(exp_v, [])), exp_v') \hookrightarrow (\Delta, \sigma, E, \text{tag}(exp_v, exp_v')) \quad (149)$$

$$(\Delta, \sigma, E, \text{iftagof } exp \text{ is } exp' \text{ then } exp'' \text{ else } exp''') \hookrightarrow (\Delta, \sigma, E \circ (\text{iftagof } [] \text{ is } exp' \text{ then } exp'' \text{ else } exp'''), exp) \quad (150)$$

$$(\Delta, \sigma, E \circ (\text{iftagof } [] \text{ is } exp' \text{ then } exp'' \text{ else } exp'''), exp_v) \hookrightarrow (\Delta, \sigma, E \circ (\text{iftagof } exp_v \text{ is } [] \text{ then } exp'' \text{ else } exp'''), exp') \quad (151)$$

$$(\Delta, \sigma, E, [lab \triangleright var = phrase, sbnds]) \hookrightarrow (\Delta, \sigma, E \circ [lab \triangleright var = [], sbnds], phrase) \quad (152)$$

$$\begin{aligned} &(\Delta, \sigma, E \circ [sbnds_v, lab \triangleright var = [], \\ &\quad lab' \triangleright var' = phrase, sbnds], val) \hookrightarrow \\ &(\Delta, \sigma, E \circ [sbnds_v, lab \triangleright var = val, \\ &\quad lab' \triangleright var' = [], \{val/var\} sbnds], \{val/var\} phrase) \end{aligned} \quad (153)$$

$$(\Delta, \sigma, E \circ [sbnds_v, lab \triangleright var = []], val) \hookrightarrow (\Delta, \sigma, E, [sbnds_v, lab \triangleright var = val]) \quad (154)$$

$$(\Delta, \sigma, E, mod.lab) \hookrightarrow (\Delta, \sigma, E \circ [].lab, mod) \quad (155)$$

$$(\Delta, \sigma, E, mod \ mod') \hookrightarrow (\Delta, \sigma, E \circ ([] \ mod'), mod) \quad (156)$$

$$(\Delta, \sigma, E \circ ([] \ mod'), mod_v) \hookrightarrow (\Delta, \sigma, E \circ (mod_v \ []), mod) \quad (157)$$

$$(\Delta, \sigma, E, mod:sig) \hookrightarrow (\Delta, \sigma, E \circ []:sig, mod) \quad (158)$$

4.2.2 Reduction

$$\begin{aligned} &(\Delta, \sigma, E \circ (exp_v^j \ []), exp_v) \hookrightarrow \\ &(\Delta, \sigma, E, \{exp_v^1/var'_1\} \cdots \{exp_v^n/var'_n\} \{exp_v/var_j\} exp_j) \\ &\text{where } \forall k \in 1..n : \\ &exp_v^k = \langle \partial \rangle \pi_{\bar{k}} \text{fix } (var'_i (var_i:con_i):con'_i \mapsto exp_i)_{i=1}^n \text{end} \end{aligned} \quad (159)$$

$$(\Delta, \sigma, E \circ (\pi_{lab} \ []), \{rbnds_v, lab=exp_v, rbnds_v'\}) \hookrightarrow (\Delta, \sigma, E, exp_v) \quad (160)$$

$$(\Delta, \sigma, E \circ (\text{handle } [] \text{ with } exp), exp_v) \hookrightarrow (\Delta, \sigma, E, exp_v) \quad (161)$$

$$(\Delta, \sigma, E \circ (\text{ref}^{con} []), exp_v) \hookrightarrow (\Delta[loc:con], \sigma[loc \mapsto exp_v], E, loc) \quad \text{if } loc \notin BV(\Delta) \quad (162)$$

$$(\Delta, \sigma, E \circ (\text{get} []), loc) \hookrightarrow (\Delta, \sigma, E, exp_v) \quad \text{if } \sigma(loc) = exp_v \quad (163)$$

$$(\Delta, \sigma, E \circ (\text{set}(loc, [])), exp_v) \hookrightarrow (\Delta, \sigma[loc \mapsto exp_v], E, \{\}) \quad (164)$$

$$(\Delta, \sigma, E \circ (\text{unroll} []), \text{roll}^{con'} exp_v) \hookrightarrow (\Delta, \sigma, E, exp_v) \quad (165)$$

$$(\Delta, \sigma, E, \text{new_tag}[con]) \hookrightarrow (\Delta[tag:con \text{ Tag}], \sigma, E, tag) \quad \text{if } tag \notin BV(\Delta) \quad (166)$$

$$(\Delta, \sigma, E \circ (\text{proj}_{lab'}^{con'} []), \text{inj}_{lab}^{con} exp_v) \hookrightarrow (\Delta, \sigma, E, exp_v) \quad (167)$$

$$(\Delta, \sigma, E \circ (\text{case}^{con} [] \text{ of } exp_1, \dots, exp_n \text{ end}), \text{inj}_{lab_k}^{\Sigma lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n} exp_v) \hookrightarrow (\Delta, \sigma, E, exp_k exp_v) \quad (168)$$

$$(\Delta, \sigma, E \circ (\text{iftagof tag}(tag, exp_v) \text{ is } [] \text{ then } exp'' \text{ else } exp'''), tag) \hookrightarrow (\Delta, \sigma, E, exp'' exp_v) \quad (169)$$

$$(\Delta, \sigma, E \circ (\text{iftagof tag}(tag, exp_v) \text{ is } [] \text{ then } exp'' \text{ else } exp'''), tag') \hookrightarrow (\Delta, \sigma, E, exp''') \quad \text{if } tag \neq tag' \quad (170)$$

$$(\Delta, \sigma, E \circ ((\lambda var: sig.mod) []), mod_v) \hookrightarrow (\Delta, \sigma, E, \{mod_v/var\} mod) \quad (171)$$

$$(\Delta, \sigma, E \circ []: sig, mod_v) \hookrightarrow (\Delta, \sigma, E, mod_v) \quad (172)$$

$$(\Delta, \sigma, E \circ (\text{handle} [] \text{ with } exp') \circ R, \text{raise}^{con} exp_v) \hookrightarrow (\Delta, \sigma, E, exp' exp_v) \quad (173)$$

$$(\Delta, \sigma, R, \text{raise}^{con} exp_v) \hookrightarrow (\Delta, \sigma, [], \text{raise}^{ans} exp_v) \quad (174)$$

5 External Language

5.1 Notation

As in the *Definition* optional elements are enclosed by single brackets $\langle \dots \rangle$ or double brackets $\langle\langle \dots \rangle\rangle$. For the purposes of this grammar all optional choices are completely independent.

5.2 Grammar of the Abstract Syntax

The (disjoint) base syntax classes include *scon* (syntactic constants) Γ *tyvar* (type variables) Γ *id* (core-level identifiers) Γ *tycon* (type constructors) Γ *strid* (structure identifiers) Γ and *funid* (functor identifiers). As in *The Definition of Standard ML* all but the first two have corresponding “long” forms containing a finite sequence of structure identifiers as a prefix.

```
expr ::= scon
      | longid
      | {lab1 = expr1, ..., labn = exprn}
      | let strdec in expr end
      | expr expr'
      | expr : ty
      | expr handle match
      | raise expr
      | fn match
      | expr1 = expr2

mrule ::= pat => expr
match ::= mrule
       | mrule | match

strdec ::= .
       | val (tyvar1, ..., tyvarn) pat = exp
       | val (tyvar1, ..., tyvarn) rec pat = exp
       | strdec1 strdec2
       | open longid1 ... longidn
       | exception id
       | exception id of ty
       | exception id = longid
       | local strdec1 in strdec2 end
       | type tybind
       | datatype datbind
       | datatype (tyvar1, ..., tyvarn) tycon =
           datatype (tyvar1, ..., tyvarn) longtycon
       | structure strbind
       | functor funbind
```

```

    tybind ::= (tyvar1, ..., tyvarn) tycon = ty <and tybind>
    datbind ::= (tyvar1, ..., tyvarn) tycon = conbind
               <and datbind>
    conbind ::= id <of ty> <<| conbind>>

    strexp ::= longstrid
              | struct strdec end
              | longfunid (longstrid)
              | longstrid : sigexp
              | longstrid :> sigexp
              | let strdec in strexp end

    spec ::= .
            | val id : ty
            | type typdesc
            | eqtype etypdesc
            | datatype datbind
            | datatype (tyvar1, ..., tyvarn) tycon' =
              datatype (tyvar1, ..., tyvarn) longtycon
            | exception id
            | exception id of ty
            | structure strid : sigexp
            | functor funid (strid : sigexp) : sigexp'
            | include sigexp
            | spec1 spec2
            | spec sharing type longid1 = longid2

    typdesc ::= (tyvar1, ..., tyvarn) tycon <and typdesc>
              | (tyvar1, ..., tyvarn) tycon = ty <and typdesc>

    etypdesc ::= (tyvar1, ..., tyvarn) tycon <and etypdesc>

    sigexp ::= sig spec end
              | sigexp where type (tyvar1, ..., tyvarn) longtycon = ty

    pat ::= scon
           | longid
           | -
           | pat : ty
           | longid pat
           | {lab1 = pat1, ..., labn = patn <,...>}
           | pat1 as pat2
           | ref pat

```

$$\begin{array}{lcl}
ty ::= & base & \\
& | & tyvar \\
& | & \{lab_1 : ty_1, \dots, lab_n : ty_n\} \\
& | & (ty_1, \dots, ty_n) \text{ longtycon} \\
& | & ty \rightarrow ty' \\
\\
strbind ::= & strid = strexp \langle \text{and } strbind \rangle & \\
funbind ::= & funid (strid : sigexp) = strexp \langle \text{and } funbind \rangle &
\end{array}$$

5.3 Syntactic Restrictions

- No record expression Γ record pattern Γ or record type may contain duplicate field labels. No *tyvar* may appear more than once in a single sequence.
- Any type variable occurring in a *conbind* must also appear in the enclosing *datbind*. Any type variable appearing in the *ty* of a *where* type must appear in the type variable sequence.
- No *val* Γ type Γ datatype Γ exception Γ structure Γ signature or functor *strdec* or *spec* may bind the same identifier twice; this applies also to value constructors within a *datbind*.
- In a *val rec* declaration Γ the pattern must be of the form

$$\{lab_1=id_1, \dots, lab_n=id_n\}$$

and the expression must be of the form

$$\{lab_1=fn \text{ match}_1, \dots, lab_n=fn \text{ match}_n\}.$$

The “...” EL-notation may not appear in the pattern.

6 Elaboration

6.1 Introduction

In addition to type-checking and type-reconstruction the elaborator performs the following tasks:

1. Datatypes are expanded into structures and signatures whose components include
 - an abstract type (implemented as a recursive sum type);
 - operations corresponding to datatype constructors as values (those datatype constructors which carry values are total functions while non value-carrying constructors are constants of the abstract type);
 - an “expose” operation which presents datatype values as elements of a (tagged) sum type.

The “generativity” of datatypes is handled via signature ascription; the type is made opaque and is therefore inequivalent to any previous type. The matching of datatypes in signatures reduces to the matching of substructures.

2. Polymorphism (including equality polymorphism) is encoded as a use of the modules system. Polymorphic values are translated into functors which can be explicitly instantiated with structures of types (and equality functions). More precisely the functor takes a structure containing type constructors of kind Ω (for types which the polymorphic value requires to be equality types the structure also contains equality functions for the instantiating types); the functor returns a structure whose single component (with label “it”) is the polymorphic value made monomorphic by instantiating it with the given types.
3. Equality polymorphism as just mentioned and equality types are handled by explicitly constructing and passing equality functions as needed. We do not explicitly distinguish types that admit equality; rather a type admits equality iff the equality compiler can create an equality function for this type.
4. We make explicit the propagation of abstract types defined locally to module-level `let` and `local` constructors or hidden through the use of transparent signature ascription. In particular the “hiding” effect of these constructs on types is implemented as *renaming* rather than simple scoping. This is necessary to obtain the same type propagation behavior as the stamp-based semantics of *The Definition*.
5. Patterns are expanded into uses of the appropriate record projections and datatype deconstructors. Thus the elaboration specifies a reference pattern compiler.
6. Each series of external language bindings (*strdec*) elaborates into a structure containing a component for every variable bound in the external language. External language *identifiers* correspond to internal language *labels*.

7. Some structure labels are explicitly marked with an asterisk (*lab**). This indicates that the structure is “open” for the purposes of identifier lookup. (See the lookup rules for more details.)
8. All coercive aspects of the signature matching relation (the reordering and forgetting of components) are handled by introducing explicit coercion functors witnessing the relation. This makes the order and number of components of a structure apparent from its signature though there is a run-time cost to signature ascription.

6.2 Notation

- The overbar function $\overline{}$ maps each EL identifier to an IL label. We assume that this function is injective, that the range is coinfinite in the set of IL labels, and that identifiers of different classes map to different labels. In particular, we assume that the parser distinguishes between the classes of expression variables, type constructors, type variables, structure identifiers, signature identifiers, and functor identifiers. However, we do not distinguish between an identifier being used as an expression variable, datatype constructor, or exception constructor.

The distinguished labels “eq”, “expose”, “it”, and “tag” used by the elaborator are not in the range of the overbar mapping, and other labels chosen to be fresh are similarly not in the range of the mapping.

We extend the overbar mapping component-wise to long identifiers, which thus map to sequences of labels.

- Optional elements are enclosed in single or double angle brackets. For each rule, either all or none of the elements in single angle brackets must be present, and similarly all or none of the elements in double angle brackets must be present. Single and double angle brackets in the same rule represent two *independent* choices.
- In some cases, the optional element notation is insufficient. Therefore, we have the additional notation

$$\left\{ \begin{array}{c} element_1 \\ \text{or} \\ element_2 \end{array} \right\}$$

which means that either $element_1$ or $element_2$ must be present. If there are multiple such choices in a single rule, this means that either the first element should always be chosen in all cases, or the second element must be chosen in all cases.

An extension of this notation gives the choices subscripts. Then all choices with the same subscript must agree (all first element or all second element) but two choices with different subscripts are completely independent.

- The elaboration maintains an elaboration context Γ which is simply a list of structure declarations (*sdecls*) except that we allow duplicate labels. When Γ appears in an IL judgment where *decls* is expected, there is an implicit coercion which drops all top-level

labels and all signature declarations. We extend the notion of variable bindings from Section 2.5 with the following:

<i>Function</i>	<i>Definition</i>
$BV(\Gamma)$	$BV(sdec, \Gamma) = \{BV(sdec)\} \cup BV(\Gamma)$
$\text{dom}(\Gamma)$	$\text{dom}(sdec, \Gamma) = \{\text{dom}(sdec)\} \cup \text{dom}(\Gamma)$

<i>Binding Phrase</i>	<i>Bound Vars</i>	<i>Scope</i>
$sdec, \Gamma$	$BV(sdec)$	Γ

- We use an operation of “syntactic concatenation with renaming” for $sbnds$ and Γ in parallel Γ for $sdec$ s. This is defined by:

$$\begin{aligned}
(\cdot ++ sbnds') : (\cdot ++ sdec') &:= sbnds' : sdec' \\
(lab^{(*)} \triangleright bnd, sbnds ++ sbnds') : (lab^{(*)} \triangleright bnd, sdec ++ sdec') &:= \\
\begin{cases} lab^{(*)} \triangleright bnd, sbnds'' : lab^{(*)} \triangleright dec, sdec'' & \text{if } lab^{(*)} \notin \text{dom}(sbnds'') \\ lab^{(*)} \triangleright bnd, sbnds'' : lab^{(*)} \triangleright dec, sdec'' & \text{otherwise} \end{cases} &\text{where } lab^{(*)} \notin \text{dom}(sbnds'') \\
\text{where } sbnds ++ sbnds' : sdec ++ sdec' &= sbnds'' : sdec''
\end{aligned}$$

6.3 Initial Basis

The elaborator assumes the presence of a structure $basis: sig_{basis}$ serving as the initial basis for the internal language. It must contain at least the following fields which define three exceptions:

$$\begin{aligned}
\overline{\text{Bind}}^* &: [\text{tag}: \text{Unit Tag}, \overline{\text{Bind}}: \text{Tagged}], \\
\overline{\text{Match}}^* &: [\text{tag}: \text{Unit Tag}, \overline{\text{Match}}: \text{Tagged}], \\
\text{fail}^* &: [\text{tag}: \text{Unit Tag}, \text{fail}: \text{Tagged}]].
\end{aligned}$$

6.4 Derived Forms

The translation also makes use of a number of derived forms of kinds Γ constructors Γ and expressions. These are shown in Figure 6.

The representation of tuples as records with numbered fields is copied from SML. The encoding of multi-argument functions as single-argument functions is also very standard Γ as is the encoding of booleans as a sum type.

The purpose of the `catch` form is to serve as a special handler for the “fail” exception Γ and to propagate all other exceptions. The fail exception is only used by the elaborator to signal a failure in pattern-matching.

$knd_1 \times \dots \times knd_n$	$\mapsto \{1:knd_1, \dots, n:knd_n\}$
knd^n	$\mapsto \{1:knd, \dots, n:knd\}$
Unit	$\mapsto \{\}$
$\text{Bool}_{\langle lab \rangle}$	$\mapsto \Sigma_{\langle lab \rangle} (\bar{1} \mapsto \text{Unit}, \bar{2} \mapsto \text{Unit})$
$con_1 \times \dots \times con_n$	$\mapsto \{\bar{1}=con_1, \dots, \bar{n}=con_n\}$
$\lambda(var_1, \dots, var_n).con$	$\mapsto \lambda var:\Omega^n.(\{\pi_1 var/var_1\} \dots \{\pi_n var/var_n\} con)$
(con_1, \dots, con_n)	$\mapsto \{1=con_1, \dots, n=con_n\}$
(exp_1, \dots, exp_n)	$\mapsto \{\bar{1}=exp_1, \dots, \bar{n}=exp_n\}$
$\lambda(var:con):con'.exp$	$\mapsto \pi_{\bar{1}} \text{fix } var'(var:con):con' \mapsto exp \text{ end}$ $var' \notin \text{FV}(exp)$
$\lambda(var_1:con_1, \dots, var_n:con_n):con.exp$	$\mapsto \lambda(var:con_1 \times \dots \times con_n):con.$ $\{\pi_1 var/var_1\} \dots \{\pi_n var/var_n\} exp$ $var \notin \text{FV}(exp)$
let bnd_1, \dots, bnd_n in exp end	$\mapsto [1=bnd_1, \dots, n=bnd_n, (n+1)=exp].(n+1)$
fail ^{con}	$\mapsto \text{raise}^{con} \text{basis.fail}^*.\text{fail}$
catch ^{con} exp with exp'	$\mapsto \text{handle } exp \text{ with } (\lambda var:\text{Tagged}.$ $\text{iftagof } var \text{ is basis.fail}^*.\text{tag then } \lambda var:\text{Unit}.exp' \text{ else raise}^{con} var$ $var \notin \text{FV}(exp'))$
$\text{pproj}_{lab_i}^{\Sigma con_1, \dots, con_n} (exp, exp')$	$\mapsto \text{case}^{con'} exp \text{ of}$ $\lambda var:con'_{lab_1}.\text{raise}^{con_i} exp', \dots,$ $\lambda var:con'_{lab_i}.\text{proj}_{lab_i}^{con_{lab_i}} var, \dots$ $\lambda var:con'_{lab_n}.\text{raise}^{con_i} exp' \text{ end}$ where $con'_{\langle lab \rangle} = \Sigma_{\langle lab \rangle} (lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n)$
false	$\mapsto \text{inj}_1^{\text{Bool}} \{\}$
true	$\mapsto \text{inj}_2^{\text{Bool}} \{\}$
if exp_1 then exp_2 else exp_3	$\mapsto \text{case}^{\text{Bool}} exp_1 \text{ of } \lambda var:\text{Bool}_{\bar{1}}.exp_3, \lambda var:\text{Bool}_{\bar{2}}.exp_2 \text{ end}$ $var \notin \text{FV}(exp_2, exp_3)$
exp_1 and exp_2	$\mapsto \text{if } exp_1 \text{ then } exp_2 \text{ else false}$
$\Lambda(var_1^{(eq)_1}, \dots, var_n^{(eq)_n}).exp$	$\mapsto \lambda(var:[1^*:[1 \triangleright var_1:\Omega\langle, eq:var_1 \times var_1 \rightarrow \text{Bool}\rangle_1], \dots,$ $n^*:[n \triangleright var_n:\Omega\langle, eq:var_n \times var_n \rightarrow \text{Bool}\rangle_n]])$ $[it=\{var.1^*.1/var_1\} \dots \{var.n^*.n/var_n\} exp]$
$\forall(var_1^{(eq)_1}, \dots, var_n^{(eq)_n}).con$	$\mapsto (var:[1^*:[1 \triangleright var_1:\Omega\langle, eq:var_1 \times var_1 \rightarrow \text{Bool}\rangle_1], \dots,$ $n^*:[n \triangleright var_n:\Omega\langle, eq:var_n \times var_n \rightarrow \text{Bool}\rangle_n]]) \rightarrow$ $[it:\{var.1^*.1/var_1\} \dots \{var.n^*.n/var_n\} con]$

Figure 6: Derived Forms

6.5 Judgment Forms

<i>Judgment...</i>	<i>Meaning...</i>
$\Gamma \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con}$	expression
$\Gamma \vdash \text{match} \rightsquigarrow \text{exp} : \text{con}$	pattern match
$\Gamma \vdash \text{strdec} \rightsquigarrow \text{sbnds} : \text{sdecs}$	declaration
$\Gamma \vdash \text{strexpr} \rightsquigarrow \text{mod} : \text{sig}$	structure expression
$\Gamma \vdash \text{spec} \rightsquigarrow \text{sdecs}$	signature specification
$\Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig} : \text{Sig}$	signature expression
$\Gamma \vdash \text{ty} \rightsquigarrow \text{con} : \Omega$	type expression
$\Gamma \vdash \text{tybind} \rightsquigarrow \text{sbnds} : \text{sdecs}$	type definition
$\Gamma \vdash \text{datbind} \rightsquigarrow \text{sbnds} : \text{sdecs}$	datatype definition
$\Gamma \vdash_{\text{ctx}} \text{labs} \rightsquigarrow \text{path} : \text{class}$	lookup in Γ
$\Gamma \vdash_{\text{ctx}} \text{labs} \rightsquigarrow \text{path}$	
$\text{decs}; \text{path} : \text{sig} \vdash_{\text{sig}} \text{labs} \rightsquigarrow \text{labs}' : \text{class}$	lookup in signature
$\text{sig} \vdash_{\text{sig}} \text{lab} \rightsquigarrow \text{labs}'$	
$\text{decs} \vdash_{\text{inst}} \rightsquigarrow [\text{sbnds}_v] : [\text{sdecs}']$	polymorphic instantiation
$\Gamma \vdash \text{pat} \Leftarrow \text{exp} : \text{con} \text{ else } \text{exp} \rightsquigarrow \text{sbnds} : \text{sbnds}$	pattern compilation
$\text{decs} \vdash_{\text{eq}} \text{con} \rightsquigarrow \text{exp}_v$	equality compilation
$\text{decs} \vdash_{\text{sub}} \text{path} : \text{sig}_0 \preceq \text{sig} \rightsquigarrow \text{mod} : \text{sig}'$	coercion compilation
$\text{decs}; \text{path} : \text{sig}_0 \vdash_{\text{sub}} \text{sdec} \rightsquigarrow \text{sbnd} : \text{sdec}'$	
$\text{sig} \vdash_{\text{wt}} \text{labs} := \text{con} : \text{knd} \rightsquigarrow \text{sig}' : \text{Sig}$	impose definition
$\text{sig} \vdash_{\text{sh}} \text{labs} := \text{labs}' : \text{knd} \rightsquigarrow \text{sig}' : \text{Sig}$	impose sharing

6.6 Translation Rules

6.6.1 Expressions

$$\boxed{\Gamma \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con}}$$

$$\frac{}{\Gamma \vdash \text{scon} \rightsquigarrow \text{scon} : \text{type}(\text{scon})} \quad (175)$$

Rule 175: We assume a meta-level function type which gives the IL type of each constant.

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path} : \text{con} \quad \text{Rule 177 does not apply.}}{\Gamma \vdash \text{longid} \rightsquigarrow \text{path} : \text{con}} \quad (176)$$

Rule 176: Monomorphic variables.

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path} : \text{con} \rightarrow \text{con}'}{\Gamma \vdash \text{longid} \rightsquigarrow \partial(\text{path}) : \text{con} \rightarrow \text{con}'} \quad (177)$$

Rule 176: Monomorphic datatype or exception constructors. Because the IL has no subsumption, these must be coerced to a partial function type when used as function values (not immediately applied).

$$\frac{\begin{array}{c} \Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path} : \text{sig} \rightarrow [\text{it} : \text{con}] \\ \Gamma \vdash_{\text{inst}} \rightsquigarrow \text{mod} : \text{sig} \\ \text{Rule 179 does not apply.} \end{array}}{\Gamma \vdash \text{longid} \rightsquigarrow \text{path}(\text{mod}).\text{it} : \text{con}} \quad (178)$$

Rule 178: Instantiation of polymorphic variables. All polymorphic functions are translated into total functors whose body contains a single component with the label "it". The module *mod* is the structure of types (and equality functions, if needed) that are used to instantiate the polymorphic function.

$$\frac{\begin{array}{c} \Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path} : \text{sig} \rightarrow [\text{it} : \text{con} \rightarrow \text{con}'] \\ \Gamma \vdash_{\text{inst}} \rightsquigarrow \text{mod} : \text{sig} \end{array}}{\Gamma \vdash \text{longid} \rightsquigarrow \partial(\text{path}(\text{mod}).\text{it}) : \text{con} \rightarrow \text{con}'} \quad (179)$$

Rule 179: Instantiation of polymorphic datatype constructors. As in Rule 176, we coerce to a partial function type.

$$\frac{\begin{array}{c} \sigma \text{ a permutation of } 1..n \\ \text{var}_1, \dots, \text{var}_n \notin \text{BV}(\Gamma) \\ \text{lab}_{\sigma(1)} < \dots < \text{lab}_{\sigma(n)} \\ \Gamma \vdash \text{expr}_1 \rightsquigarrow \text{exp}_1 : \text{con}_1 \dots \Gamma \vdash \text{expr}_n \rightsquigarrow \text{exp}_n : \text{con}_n \end{array}}{\Gamma \vdash \{ \text{lab}_1 = \text{expr}_1, \dots, \text{lab}_n = \text{expr}_n \} \rightsquigarrow \text{let } \text{var}_1 = \text{exp}_1, \dots, \text{var}_n = \text{exp}_n \text{ in } \{ \overline{\text{lab}_{\sigma(1)}} = \text{var}_{\sigma(1)}, \dots, \overline{\text{lab}_{\sigma(n)}} = \text{var}_{\sigma(n)} \} \text{ end} : \{ \overline{\text{lab}_{\sigma(1)}} : \text{con}_{\sigma(1)}, \dots, \overline{\text{lab}_{\sigma(n)}} : \text{con}_{\sigma(n)} \}} \quad (180)$$

Rule 180: The order in which labels appear in the record type is significant for the IL but not the EL, so in the translation we normalize record values and types by sorting the labels with some fixed ordering $<$. The order in which effects occur, however, is determined by the order of the expressions in the EL.

$$\frac{\begin{array}{c} \Gamma \vdash \text{strdec} \rightsquigarrow \text{sbinds} : \text{sdecs} \\ \text{var} \notin \text{BV}(\Gamma) \quad \Gamma, 1^* \triangleright \text{var} : [\text{sdecs}] \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con}' \\ \Gamma, \text{var} : [\text{sdecs}] \vdash \text{con}' \equiv \text{con} : \Omega \quad \Gamma \vdash \text{con} : \Omega \end{array}}{\Gamma \vdash \text{let } \text{strdec} \text{ in } \text{expr} \text{ end} \rightsquigarrow \text{let } \text{var} = [\text{mod}] \text{ in } \text{exp} \text{ end} : \text{con}} \quad (181)$$

Rule 181: The “starred label” convention is used here to make the locally-defined bindings accessible while translating *expr*. The elaborator verifies that the translated expression can be given a type, which must not depend on any abstract types (e.g., datatypes) defined in *strdec*.

$$\frac{\begin{array}{c} \Gamma \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con}'' \rightarrow \text{con} \\ \Gamma \vdash \text{expr}' \rightsquigarrow \text{exp}' : \text{con}' \\ \Gamma \vdash \text{con}' \equiv \text{con}'' : \Omega \end{array}}{\Gamma \vdash \text{expr expr}' \rightsquigarrow \text{exp exp}' : \text{con}} \quad (182)$$

Rule 182: General application.

$$\frac{\begin{array}{c} \Gamma \vdash_{\text{ctx}} \text{longid} \rightsquigarrow \text{path} : \text{con}' \rightarrow \text{con} \\ \Gamma \vdash \text{expr}' \rightsquigarrow \text{exp}' : \text{con}' \end{array}}{\Gamma \vdash \text{longid expr}' \rightsquigarrow \text{exp exp}' : \text{con}} \quad (183)$$

Rule 183: Application of a monomorphic datatype constructor, which is valuable if *expr'* is valuable.

$$\frac{\begin{array}{c} \Gamma \vdash_{\text{ctx}} \text{longid} \rightsquigarrow \text{path} : \text{sig} \rightarrow [\text{it} : \text{con}' \rightarrow \text{con}] \\ \Gamma \vdash_{\text{inst}} \rightsquigarrow \text{mod} : \text{sig} \\ \Gamma \vdash \text{expr}' \rightsquigarrow \text{exp}' : \text{con}' \end{array}}{\Gamma \vdash \text{longid expr}' \rightsquigarrow (\text{path}(\text{mod}).\text{it}) \text{exp}' : \text{con}} \quad (184)$$

Rule 184: Application of a polymorphic datatype constructor, which is valuable if *expr'* is valuable.

$$\frac{\begin{array}{c} \Gamma \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con} \\ \Gamma \vdash \text{ty} \rightsquigarrow \text{con}' : \Omega \quad \Gamma \vdash \text{con} \equiv \text{con}' : \Omega \end{array}}{\Gamma \vdash \text{expr} : \text{ty} \rightsquigarrow \text{exp} : \text{con}} \quad (185)$$

Rule 185: Type constraints on expressions are verified, but do not appear in the translation.

$$\frac{\begin{array}{c} \Gamma \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con} \\ \Gamma \vdash \text{match} \rightsquigarrow \text{exp}' : \text{Tagged} \rightarrow \text{con}' \\ \Gamma \vdash \text{con} \equiv \text{con}' : \Omega \\ \text{var} \notin \text{BV}(\Gamma) \end{array}}{\Gamma \vdash \text{expr handle match} \rightsquigarrow \text{handle exp with} \lambda(\text{var} : \text{Tagged}) : \text{con}. (\text{catch}^{\text{con}} \text{exp}' \text{var with raise}^{\text{con}} \text{var}) : \text{con}} \quad (186)$$

Rule 186: The handling expression *exp' var* will fail if the handler pattern does not match the exception caught by the IL handle, in which case we re-raise the exception.

$$\frac{\Gamma \vdash \text{expr} \rightsquigarrow \text{exp} : \text{Tagged} \quad \Gamma \vdash \text{con} : \Omega}{\Gamma \vdash \text{raise } \text{expr} \rightsquigarrow \text{raise}^{\text{con}} \text{exp} : \text{con}} \quad (187)$$

Rule 187: **raise** expressions can be given any (valid) type. To preserve the property that every IL expression has a unique type up to equivalence, we annotate the IL **raise** with this type.

$$\frac{\Gamma \vdash \text{match} \rightsquigarrow \text{exp} : \text{con}_1 \rightarrow \text{con}_2 \quad \text{var} \notin \text{BV}(\Gamma)}{\Gamma \vdash \text{fn } \text{match} \rightsquigarrow \lambda(\text{var} : \text{con}_1) : \text{con}_2. (\text{catch}^{\text{con}_2} \text{exp } \text{var} \text{ with } \text{raise}^{\text{con}_2} \text{basis.Match}^* \text{Match}) : \text{con}_1 \rightarrow \text{con}_2} \quad (188)$$

Rule 188: The application $\text{exp } \text{var}$ will fail if the match fails; we turn the failure into a match exception. The resulting function has a partial type because it can (syntactically) raise an exception.

$$\frac{\begin{array}{c} \Gamma \vdash \text{expr}_1 \rightsquigarrow \text{exp}_1 : \text{con}_1 \quad \Gamma \vdash \text{expr}_2 \rightsquigarrow \text{exp}_2 : \text{con}_2 \\ \Gamma \vdash \text{con}_1 \equiv \text{con}_2 : \Omega \quad \Gamma \vdash_{\text{eq}} \text{con}_1 \rightsquigarrow \text{exp} \end{array}}{\Gamma \vdash \text{expr}_1 = \text{expr}_2 \rightsquigarrow \text{exp}(\text{exp}_1, \text{exp}_2) : \text{Bool}} \quad (189)$$

Rule 189: Translation of equality comparison; exp is the equality function generated by the equality compiler and has type $\text{con} \times \text{con} \rightarrow \text{Bool}$.

6.6.2 Matches

$$\boxed{\Gamma \vdash \text{match} \rightsquigarrow \text{exp} : \text{con}}$$

$$\frac{\begin{array}{c} \text{var}, \text{var}' \notin \text{BV}(\Gamma) \quad \Gamma \vdash \text{con}' : \Omega \\ \Gamma \vdash \text{pat} \Leftarrow \text{var}' : \text{con}' \text{ else } \text{basis.fail}^* \text{fail} \rightsquigarrow \text{sbnds} : \text{sdecs} \\ \Gamma, 1^* \triangleright \text{var} : [\text{sdecs}] \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con} \end{array}}{\Gamma \vdash \text{pat} \Rightarrow \text{expr} \rightsquigarrow \lambda(\text{var}' : \text{con}') : \text{con}. \text{let } \text{var} = [\text{sbnds}] \text{ in } \text{exp } \text{end} : \text{con}' \rightarrow \text{con}} \quad (190)$$

Rule 190: The result of translating a match is a function that may fail if the match fails. The IL **let** expression is well-formed because the pattern compiler returns no type components in sbnds .

$$\frac{\begin{array}{c} \text{var} \notin \text{BV}(\Gamma) \\ \Gamma \vdash \text{mrule} \rightsquigarrow \text{exp} : \text{con}_1 \rightarrow \text{con}_2 \\ \Gamma \vdash \text{match} \rightsquigarrow \text{exp}' : \text{con}'_1 \rightarrow \text{con}'_2 \\ \Gamma \vdash \text{con}_1 \rightarrow \text{con}_2 \equiv \text{con}'_1 \rightarrow \text{con}'_2 : \Omega \end{array}}{\Gamma \vdash \text{mrule} \mid \text{match} \rightsquigarrow \lambda(\text{var} : \text{con}_1) : \text{con}_2. \text{catch}^{\text{con}} \text{exp } \text{var} \text{ with } \text{exp}' \text{var} : \text{con}' \rightarrow \text{con}} \quad (191)$$

Rule 191: The failure of pattern matching in the first clause is caught, and we try again with the next clause.

6.6.3 Declarations

$$\boxed{\Gamma \vdash strdec \rightsquigarrow sbnds : sdec}$$

$$\frac{\begin{array}{c} var \notin BV(\Gamma) \\ \Gamma \vdash expr \rightsquigarrow exp : con \\ \Gamma, var:con \vdash pat \Leftarrow var : con \text{ else } basis.\overline{Bind}^*.\overline{Bind} \rightsquigarrow sbnds : sdec \end{array}}{\Gamma \vdash \text{val } () \text{ pat} = expr \rightsquigarrow 1 \triangleright var = exp, sbnds : 1 \triangleright var : con, sdec} \quad (192)$$

Rule 192: Monomorphic, non-recursive variable bindings.

$$\begin{array}{l} sig_p = [\overline{\langle eq \rangle_1 tyvar_1}^* : [\overline{\langle eq \rangle_1 tyvar_1} \triangleright var : \Omega, \langle eq : var \times var \rightarrow Bool \rangle_1], \dots, \\ \quad \overline{\langle eq \rangle_n tyvar_n}^* : [\overline{\langle eq \rangle_n tyvar_n} \triangleright var : \Omega, \langle eq : var \times var \rightarrow Bool \rangle_n], \\ \quad 1^* : [1 \triangleright var : \Omega, \langle eq : var \times var \rightarrow Bool \rangle_1], \dots, \\ \quad m^* : [m \triangleright var : \Omega, \langle eq : var \times var \rightarrow Bool \rangle_{m'}]] \\ \Gamma, 1^* \triangleright var_p : sig_p \vdash expr \rightsquigarrow exp : con \\ \Gamma, 1^* \triangleright var_p : sig_p \vdash exp \downarrow con \\ \Gamma, 1^* \triangleright var_p : sig_p \vdash basis.\overline{Bind}^*.\overline{Bind} \Leftarrow exp : con \text{ else } pat \rightsquigarrow sbnd_1, \dots, sbnd_n : sdec_1, \dots, sdec_n \\ \forall i \in 1..n : \\ \quad sbnd_i = lab_i = exp_i \\ \quad sdec_i = lab_i : con_i \\ sbnd'_i = \begin{cases} lab_i = (var_p : sig_p) \rightarrow [it = exp_i] & \text{if } \Gamma, var_p : sig_p \vdash sbnd_i \downarrow sdec_i \\ lab_i = exp_i & \text{if } \Gamma \vdash exp_i : con_i \end{cases} \\ sdec'_i = \begin{cases} lab_i : (var_p : sig_p) \rightarrow [it : con_i] & \text{if } \Gamma, var_p : sig_p \vdash sbnd_i \downarrow sdec_i \\ lab_i : con_i & \text{if } \Gamma \vdash exp_i : con_i \end{cases} \\ \hline \Gamma \vdash \text{val } (\langle eq \rangle_1 tyvar_1, \dots, \langle eq \rangle_n tyvar_n) \text{ pat} = expr \rightsquigarrow \\ \quad sbnd'_1, \dots, sbnd'_n : sdec'_1, \dots, sdec'_n \end{array} \quad (193)$$

Rule 193: Polymorphic, non-recursive val bindings. We assume a prepass which annotates val declarations with the explicit type variables implicitly scoped by that declaration. Type inference may introduce m new type (or equality type) variables which not mentioned in the source (as in $\text{val } f = \text{fn } x \Rightarrow x$ or $\text{val } f = \text{fn } x \Rightarrow (x=x)$). In this formulation, some variables in the pattern may become polymorphic while others may remain monomorphic. Therefore, for each i , either $sbnd'_i$ and $sdec'_i$ must both choose the first option (polymorphic) or must both choose the second option (monomorphic).

$$\begin{aligned}
& sig_p = [\overline{\langle eq \rangle_1 tyvar_1}^* : [\overline{\langle eq \rangle_1 tyvar_1 \triangleright var} : \Omega \langle eq : var \times var \rightarrow Bool \rangle_1], \dots, \\
& \quad \overline{\langle eq \rangle_m tyvar_m}^* : [\overline{\langle eq \rangle_m tyvar_m \triangleright var} : \Omega \langle eq : var \times var \rightarrow Bool \rangle_m] \\
& \quad 1^* : [1 \triangleright var : \Omega \langle eq : var \times var \rightarrow Bool \rangle_1], \dots, \\
& \quad k^* : [k \triangleright var : \Omega \langle eq : var \times var \rightarrow Bool \rangle_{k'}]] \\
& \Gamma' = \Gamma, lab^* \triangleright var_p : sig_p, \overline{id_1} \triangleright var'_1 : con_1 \rightarrow con'_1, \dots, \overline{id_n} \triangleright var'_n : con_n \rightarrow con'_n \\
& \forall i \in 1..n : \\
& \quad \Gamma' \vdash match_i \rightsquigarrow \lambda(var_i : con_i) : con'_i.exp_i : con_i \rightarrow con'_i \\
& \quad \text{If } \Gamma \vdash_{ctx} id_i \rightsquigarrow path'' : con'' \text{ then } path''.expose, path''.tag \text{ aren't well-formed.} \\
& \quad exp = \text{fix } var'_1 (var_1 : con_1) : con'_1 \mapsto exp_1, \dots, var'_m (var_m : con_m) : con'_m \mapsto exp_m \text{ end} \\
\hline
& \Gamma \vdash \text{val } (\langle eq \rangle_1 tyvar_1, \dots, \langle eq \rangle_m tyvar_m) \text{ rec } \{(lab_i = id_i)_{i=1}^n\} = \{(lab_i = \text{fn } match_i)_{i=1}^n\} \rightsquigarrow \\
& \quad \overline{id_1} = \lambda var_p : sig_p. [it = \pi_1 exp], \dots, \overline{id_n} = \lambda var_p : sig_p. [it = \pi_n exp] : \\
& \quad \overline{id_1} : (var_p : sig_p) \rightarrow [it : con_1 \rightarrow con'_1], \dots, \overline{id_n} : (var_p : sig_p) \rightarrow [it : con_n \rightarrow con'_n]
\end{aligned} \tag{194}$$

Rule 194: This rule handles recursive **val** bindings. As in Rule 193, we assume that the implicit scoping of explicit type variables has been made explicit by a prepass over the EL. We also assume that **val rec ...** and **...** is syntactic sugar for a single **val rec** binding of a record of functions. Since this rule does not use the pattern compiler, we must explicitly check that we're do not redefine datatype or exception constructors.

$$\begin{aligned}
& \Gamma \vdash strdec_1 \rightsquigarrow sbnds_1 : sdecs_1 \\
& \Gamma, sdecs_1 \vdash strdec_2 \rightsquigarrow sbnds_2 : sdecs_2 \\
\hline
& \Gamma \vdash strdec_1 strdec_2 \rightsquigarrow sbnds_1 ++ sbnds_2 : sdecs_1 ++ sdecs_2
\end{aligned} \tag{195}$$

Rule 195: We use the syntactic-concatenation-with-renaming operation defined in Section 6.2.

$$\begin{aligned}
& \forall i \in 1..n : \quad \Gamma \vdash_{ctx} \overline{longstrid_i} \rightsquigarrow path_i : sig_i \\
\hline
& \Gamma \vdash \text{open } longstrid_1 \dots longstrid_n \rightsquigarrow \\
& \quad 1^* = path_1, \dots, n^* = path_n : 1^* : sig_1, \dots, n^* : sig_n
\end{aligned} \tag{196}$$

$$\begin{aligned}
& \Gamma \vdash ty \rightsquigarrow con : \Omega \quad var \notin BV(\Gamma) \\
\hline
& \Gamma \vdash \text{exception } id \text{ of } ty \rightsquigarrow \\
& \quad \overline{id}^* = [\text{tag} \triangleright var = \text{new_tag}[con], \overline{id} = \lambda(var' : con) : \text{Tagged.tag}(var, var')] : \\
& \quad \overline{id}^* : [\text{tag} \triangleright var : con \text{ Tag}, \overline{id} : con \rightarrow \text{Tagged}]
\end{aligned} \tag{197}$$

$$\begin{aligned}
& \Gamma \vdash \text{exception } id \rightsquigarrow \\
& \quad \overline{id}^* = [\text{tag} \triangleright var = \text{new_tag}[\text{Unit}], \overline{id} = \text{tag}(var, \{\})] : \\
& \quad \overline{id}^* : [\text{tag} \triangleright var : \text{Unit Tag}, \overline{id} : \text{Tagged}]
\end{aligned} \tag{198}$$

$$\begin{aligned}
& \Gamma \vdash_{ctx} \overline{longid} \rightsquigarrow path.lab : con \\
& \Gamma \vdash path.tag : con' \\
\hline
& \Gamma \vdash \text{exception } id = longid \rightsquigarrow \\
& \quad \overline{id}^* = [\text{tag} = path.tag, \overline{id} = path.lab] : \overline{id}^* : [\text{tag} : con', \overline{id} : con]
\end{aligned} \tag{199}$$

Rule 199: Structures containing a “tag” component are created by EL exception declarations only.

$$\begin{array}{c}
\text{var} \notin \text{BV}(\Gamma) \\
\Gamma \vdash \text{strdec}_1 \rightsquigarrow \text{sbnds}_1 : \text{sdec}_1 \\
\Gamma, 1^* \triangleright \text{var} : [\text{sdec}_1] \vdash \text{strdec}' \rightsquigarrow \text{sbnds}_2 : \text{sdec}_2 \\
\hline
\Gamma \vdash \text{local strdec in strdec}' \text{ end} \rightsquigarrow \\
1 \triangleright \text{var} = [\text{sbnds}_1], \text{sbnds}_2 : 1 \triangleright \text{var} : [\text{sdec}_1], \text{sdec}_2
\end{array} \quad (200)$$

Rule 200: We create bindings for all of the declarations, but the local bindings are segregated into a substructure inaccessible from the EL.

$$\begin{array}{c}
\Gamma \vdash \text{tybind} \rightsquigarrow \text{sbnds} : \text{sdec}_s \\
\hline
\Gamma \vdash \text{type tybind} \rightsquigarrow \text{sbnds} : \text{sdec}_s
\end{array} \quad (201)$$

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longtycon}} \rightsquigarrow \text{path}.\overline{\text{tycon}} : \Omega^m \Rightarrow \Omega \\
\Gamma \vdash \text{path} : [\overline{\text{tycon}} \triangleright \text{var} : \Omega^m \Rightarrow \Omega, \text{lab}_1 \triangleright \text{dec}_1, \dots, \text{lab}_n \triangleright \text{dec}_n] \\
\text{lab}_n = \text{expose} \\
\hline
\Gamma \vdash \text{datatype } (\text{tyvar}_1, \dots, \text{tyvar}_m) \text{ tycon}' \rightsquigarrow \\
= \text{datatype } (\text{tyvar}_1, \dots, \text{tyvar}_m) \text{ longtycon} \\
\overline{\text{tycon}'}^* = [\overline{\text{tycon}'} \triangleright \text{var} = \text{path}.\overline{\text{tycon}}, \text{lab}_1 = \text{path}.\text{lab}_1, \dots, \text{lab}_n = \text{path}.\text{lab}_n] : \\
\overline{\text{tycon}'}^* = [\overline{\text{tycon}'} \triangleright \text{var} : \Omega^m \Rightarrow \Omega = \text{path}.\overline{\text{tycon}}, \text{lab}_1 = \text{dec}_1, \dots, \text{lab}_n = \text{dec}_n]
\end{array} \quad (202)$$

Rule 202: SML '96 adds the ability to copy datatypes. Only structures which are the translations of EL datatype declarations have an initial type component and a final component named “expose”.

$$\begin{array}{c}
\Gamma \vdash \text{datbind} \rightsquigarrow \text{sbnds} : \text{sdec}_s \\
\hline
\Gamma \vdash \text{datatype datbind} \rightsquigarrow \text{sbnds} : \text{sdec}_s
\end{array} \quad (203)$$

$$\begin{array}{c}
\Gamma \vdash \text{strbind} \rightsquigarrow \text{sbnds} : \text{sdec}_s \\
\hline
\Gamma \vdash \text{structure strbind} \rightsquigarrow \text{sbnds} : \text{sdec}_s
\end{array} \quad (204)$$

$$\begin{array}{c}
\Gamma \vdash \text{funbind} \rightsquigarrow \text{sbnds} : \text{sdec}_s \\
\hline
\Gamma \vdash \text{functor funbind} \rightsquigarrow \text{sbnds} : \text{sdec}_s
\end{array} \quad (205)$$

6.6.4 Structure Expressions

$$\boxed{\Gamma \vdash \text{strexpr} \rightsquigarrow \text{mod} : \text{sig}}$$

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longstrid}} \rightsquigarrow \text{path} : \text{sig} \\
\hline
\Gamma \vdash \text{longstrid} \rightsquigarrow \text{path} : \text{sig}
\end{array} \quad (206)$$

$$\begin{array}{c}
\Gamma \vdash \text{strdec} \rightsquigarrow \text{sbnds} : \text{sdec}_s \\
\hline
\Gamma \vdash \text{struct strdec end} \rightsquigarrow \text{sbnds} : \text{sdec}_s
\end{array} \quad (207)$$

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longfunid}} \rightsquigarrow \text{path}_f : (var_1 : sig_1) \rightarrow sig_2 \\
\Gamma \vdash_{\text{ctx}} \overline{\text{longstrid}} \rightsquigarrow \text{path} : sig \\
\Gamma \vdash_{\text{sub}} \text{path} : sig \preceq sig_1 \rightsquigarrow \text{mod} : sig' \\
\Gamma \vdash (var_1 : sig) \rightarrow sig_2 \equiv sig' \rightarrow sig'' : \text{Sig} \\
\hline
\Gamma \vdash \text{longfunid}(\text{longstrid}) \rightsquigarrow (\text{path}_f : sig' \rightarrow sig'') \text{mod} : sig''
\end{array} \tag{208}$$

Rule 208: We insert an explicit coercion to drop and reorder components of the argument structure (which has signature sig), in order to match the domain signature of the functor (sig_1). The signature sig' is the most-specific (and fully transparent) signature of the coerced structure, which may expose more types (is a sub-signature of) sig_1 .

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \text{longstrid} \rightsquigarrow \text{path} : sig \quad \Gamma \vdash \text{sigexp} \rightsquigarrow sig' : \text{Sig} \\
\Gamma \vdash_{\text{sub}} \text{path} : sig \preceq sig' \rightsquigarrow \text{mod} : sig'' \\
\hline
\Gamma \vdash \text{longstrid} : \text{sigexp} \rightsquigarrow \text{mod} : sig''
\end{array} \tag{209}$$

Rule 209: As in SML, ascribing a signature to a structure using “:” hides components (this hiding being accomplished here via an explicit coercion), but allows the identity of the remaining type components to leak through. The rules for coercions ensure that sig'' will be fully transparent, maximizing propagation of type information.

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \text{longstrid} \rightsquigarrow \text{path} : sig \quad \Gamma \vdash \text{sigexp} \rightsquigarrow sig' : \text{Sig} \\
\Gamma \vdash_{\text{sub}} \text{path} : sig \preceq sig' \rightsquigarrow \text{mod} : sig'' \\
\hline
\Gamma \vdash \text{longstrid} : \text{sigexp} \rightsquigarrow (\text{mod} : sig') : sig'
\end{array} \tag{210}$$

Rule 210: Ascribing a signature to a structure with $:>$ not only hides components, but restricts information about types to that which appears in the signature.

$$\begin{array}{c}
var \notin \text{BV}(\Gamma) \\
\Gamma \vdash \text{strdec} \rightsquigarrow sbnds : sdecs \\
\Gamma, 1^* \triangleright var : [sdecs] \vdash \text{strexpr} \rightsquigarrow \text{mod} : sig \\
\hline
\Gamma \vdash \text{let strdec in strexp end} \rightsquigarrow \\
[1 \triangleright var = [sbnds], 2^* = \text{mod}] : [1 \triangleright var : [sdecs], 2^* : sig]
\end{array} \tag{211}$$

6.6.5 Structure Bindings

$$\boxed{\Gamma \vdash \text{strbind} \rightsquigarrow sbnds : sdecs}$$

$$\begin{array}{c}
\Gamma \vdash \text{strexpr}_1 \rightsquigarrow \text{mod}_1 : sig_1 \quad \langle \dots \quad \Gamma \vdash \text{strexpr}_n \rightsquigarrow \text{mod}_n : sig_n \rangle \\
\Gamma \vdash \text{strid}_1 = \text{strexpr}_1 \langle \text{and } \dots \text{ and } \text{strid}_n = \text{strexpr}_n \rangle \rightsquigarrow \\
\overline{\text{strid}_1 = \text{mod}_1 \langle \dots, \text{strid}_n = \text{mod}_n \rangle : \text{strid}_1 : sig_1 \langle \dots, \text{strid}_n : sig_n \rangle}
\end{array} \tag{212}$$

6.6.6 Functor Bindings

$$\boxed{\Gamma \vdash \text{funbind} \rightsquigarrow \text{sbnds} : \text{sdecs}}$$

$$\frac{\begin{array}{c} \text{var} \notin \text{BV}(\Gamma) \\ \Gamma \vdash \text{sigexp}_1 \rightsquigarrow \text{sig}_1 : \text{Sig} \quad \Gamma, \overline{\text{strid}_1} \triangleright \text{var} : \text{sig}_1 \vdash \text{strex}_1 \rightsquigarrow \text{mod}_1 : \text{sig}'_1 \\ \langle \cdot \rangle \\ \langle \Gamma \vdash \text{sigexp}_n \rightsquigarrow \text{sig}_n : \text{Sig} \quad \Gamma, \overline{\text{strid}_n} \triangleright \text{var} : \text{sig}_n \vdash \text{strex}_n \rightsquigarrow \text{mod}_n : \text{sig}'_n \rangle \end{array}}{\begin{array}{c} \Gamma \vdash \text{funid}_1(\text{strid}_1 : \text{sigexp}_1) = \text{strex}_1 \\ \langle \text{and} \dots \text{and funid}_n(\text{strid}_n : \text{sigexp}_1) = \text{strex}_n \rangle \rightsquigarrow \\ (\text{funid}_1 = \lambda \text{var} : \text{sig}_1. \text{mod}_1 : \text{var} : \text{sig}_1 \rightarrow \text{sig}'_1)_{i=1}^n : \\ (\text{funid}_1 : \text{var} : \text{sig}_1 \rightarrow \text{sig}'_1)_{i=1}^n \end{array}} \quad (213)$$

Rule 213: As for functions at the expression level, user-defined functors are given partial functor types.

6.6.7 Specifications

$$\boxed{\Gamma \vdash \text{spec} \rightsquigarrow \text{sdecs}}$$

$$\overline{\Gamma \vdash \cdot \rightsquigarrow \cdot} \quad (214)$$

$$\frac{\begin{array}{c} \text{TV}(\text{ty}) = \emptyset \\ \Gamma \vdash \text{ty} \rightsquigarrow \text{con} : \Omega \end{array}}{\Gamma \vdash \text{val id} : \text{ty} \rightsquigarrow \overline{\text{id}} : \text{con}} \quad (215)$$

Rule 215: A value specification is monomorphic if the EL type contains no type variables. (The set of type variables of the EL type ty is denoted by $\text{TV}(\text{ty})$.)

$$\frac{\begin{array}{c} \text{TV}(\text{ty}) = \{\langle \text{eq} \rangle_1 \text{tyvar}_1, \dots, \langle \text{eq} \rangle_n \text{tyvar}_n\} \neq \emptyset \\ \text{sig}_p = \frac{\langle \text{eq} \rangle_1 \text{tyvar}_1^* : [\langle \text{eq} \rangle_1 \text{tyvar}_1 \triangleright \text{var} : \Omega \langle \text{eq} : \text{var} \times \text{var} \rightarrow \text{Bool} \rangle_1], \dots, \langle \text{eq} \rangle_n \text{tyvar}_n^* : [\langle \text{eq} \rangle_n \text{tyvar}_n \triangleright \text{var} : \Omega \langle \text{eq} : \text{var} \times \text{var} \rightarrow \text{Bool} \rangle_n]}{\Gamma, \text{lab}^* \triangleright \text{var} : \text{sig}_p \vdash \text{ty} \rightsquigarrow \text{con} : \Omega} \end{array}}{\Gamma \vdash \text{val id} : \text{ty} \rightsquigarrow \overline{\text{id}} : (\text{var} : \text{sig}_p) \rightarrow [\text{it} : \text{con}]} \quad (216)$$

Rule 216: Specification of a polymorphic value.

$$\frac{\Gamma \vdash \text{typdesc} \rightsquigarrow \text{sdecs}}{\Gamma \vdash \text{type typdesc} \rightsquigarrow \text{sdecs}} \quad (217)$$

$$\frac{\Gamma \vdash \text{etypdesc} \rightsquigarrow \text{sdecs}}{\Gamma \vdash \text{eqtype etypdesc} \rightsquigarrow \text{sdecs}} \quad (218)$$

$$\frac{\Gamma \vdash \text{ty} \rightsquigarrow \text{con} : \Omega}{\Gamma \vdash \text{exception id of ty} \rightsquigarrow \overline{\text{id}} : [\text{tag} : \text{con Tag}, \overline{\text{id}} : \text{con} \rightarrow \text{Tagged}]} \quad (219)$$

$$\frac{}{\Gamma \vdash \text{exception } id \rightsquigarrow \overline{id} : [\text{tag} : \text{con Tag}, \overline{id} : \text{Tagged}]} \quad (220)$$

$$\frac{\Gamma \vdash \text{datbind} \rightsquigarrow \text{sbind} : 1^* : [\text{sdecs}]}{\Gamma \vdash \text{datatype } \text{datbind} \rightsquigarrow \text{sdecs}} \quad (221)$$

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{\text{longtycon}} \rightsquigarrow \text{path}.\overline{\text{tycon}} : \Omega^n \Rightarrow \Omega \quad \Gamma \vdash \text{path} : [\overline{\text{tycon}} : \Omega^n \Rightarrow \Omega, \text{sdecs}]}{\Gamma \vdash \text{datatype } (tyvar_1, \dots, tyvar_n) \text{ tycon}' \rightsquigarrow \text{datatype } (tyvar_1, \dots, tyvar_n) \overline{\text{longtycon}} \text{ tycon}'^* : [\text{tycon}' \triangleright \text{var} : \Omega^n \Rightarrow \Omega = \text{path}.\overline{\text{tycon}}, \text{sdecs}]} \quad (222)$$

$$\frac{\Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig} : \text{Sig}}{\Gamma \vdash \text{structure } \text{strid} : \text{sigexp} \rightsquigarrow \overline{\text{strid}} : \text{sig}} \quad (223)$$

$$\frac{\Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig} : \text{Sig} \quad \text{var} \notin \text{BV}(\Gamma) \quad \Gamma, \overline{\text{strid}} \triangleright \text{var} : \text{sig} \vdash \text{sigexp}' \rightsquigarrow \text{sig}' : \text{Sig}}{\Gamma \vdash \text{functor } \text{funid } (\text{strid} : \text{sigexp}) : \text{sigexp}' \rightsquigarrow \overline{\text{funid}} : (\text{var} : \text{sig} \rightarrow \text{sig}')} \quad (224)$$

$$\frac{\Gamma \vdash \text{sigexp} \rightsquigarrow [\text{sdecs}] : \text{Sig}}{\Gamma \vdash \text{include } \text{sigexp} \rightsquigarrow \text{sdecs}} \quad (225)$$

$$\frac{\Gamma \vdash \text{spec}_1 \rightsquigarrow \text{sdecs}_1 \quad \Gamma, \text{sdecs}_1 \vdash \text{spec}_2 \rightsquigarrow \text{sdecs}_2 \quad \Gamma \vdash \text{sdecs}_1, \text{sdecs}_2 \text{ ok}}{\Gamma \vdash \text{spec}_1 \text{ spec}_2 \rightsquigarrow \text{sdecs}_1, \text{sdecs}_2} \quad (226)$$

Rule 226: We disallow redeclaring EL identifiers in a signature. In the presence of `include`, we cannot syntactically restrict the EL to guarantee the syntactic concatenation of sdecs_1 and sdecs_2 will be well-formed.

$$\frac{\begin{array}{c} \Gamma \vdash \text{spec} \rightsquigarrow \text{sig} : \text{Sig} \\ \text{var} \notin \text{BV}(\Gamma) \\ \Gamma; \text{var} : \text{sig} \vdash_{\text{sig}} \overline{\text{longid}_1} \rightsquigarrow \text{labs}_1 : \text{knd} \\ \Gamma; \text{var} : \text{sig} \vdash_{\text{sig}} \overline{\text{longid}_2} \rightsquigarrow \text{labs}_2 : \text{knd} \\ \Gamma, \text{var} : \text{sig} \vdash \text{var}.\text{labs}_1 \equiv \text{var}.\text{labs}'_1 : \text{knd} \\ \Gamma, \text{var} : \text{sig} \vdash \text{var}.\text{labs}_2 \equiv \text{var}.\text{labs}'_2 : \text{knd} \\ \left\{ \begin{array}{l} \text{sig} \vdash_{\text{sh}} \text{labs}'_1 := \text{labs}'_2 : \text{knd} \rightsquigarrow \text{sig}' : \text{Sig} \\ \text{or} \\ \text{sig} \vdash_{\text{sh}} \text{labs}'_2 := \text{labs}'_1 : \text{knd} \rightsquigarrow \text{sig}' : \text{Sig} \end{array} \right\} \end{array}}{\Gamma \vdash \text{spec sharing type } \overline{\text{longid}_1} = \overline{\text{longid}_2} \rightsquigarrow \text{sig}' : \text{Sig}} \quad (227)$$

Rule 227: A type component in a signature is considered abstract, and hence eligible to appear in a sharing constraint, if it is equivalent to an opaque type (type component that is not a type abbreviation) in the *spec*.

We find the two opaque types to which the given components are equivalent, and patch the signature such that the opaque type with the smaller scope becomes a type abbreviation for the other opaque type.

6.6.8 Signature Expressions

$$\boxed{\Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig} : \text{Sig}}$$

$$\frac{\Gamma \vdash \text{spec} \rightsquigarrow \text{sig} : \text{Sig}}{\Gamma \vdash \text{sig spec end} \rightsquigarrow \text{sig} : \text{Sig}} \quad (228)$$

$$\frac{\begin{array}{c} \text{var}, \text{var}_1, \dots, \text{var}_n \notin \text{BV}(\Gamma) \\ \Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig} : \text{Sig} \\ \Gamma, \overline{\text{tyvar}_1 \triangleright \text{var}_1 : \Omega}, \dots, \overline{\text{tyvar}_n \triangleright \text{var}_n : \Omega} \vdash \text{ty} \rightsquigarrow \text{con} : \Omega \\ \Gamma; \text{var} : \text{sig} \vdash_{\text{sig}} \text{longtycon} \rightsquigarrow \text{labs} : \Omega^n \Rightarrow \Omega \\ \Gamma, \text{var} : \text{sig} \vdash \text{var.labs} \equiv \text{var.labs}' : \Omega^n \Rightarrow \Omega \\ \text{sig} \vdash_{\text{wt}} \text{labs}' := \lambda(\text{var}_1, \dots, \text{var}_n). \text{con} : \Omega^n \Rightarrow \Omega \rightsquigarrow \text{sig}' : \text{Sig} \end{array}}{\Gamma \vdash \text{sigexp where type } (\text{tyvar}_1, \dots, \text{tyvar}_n) \text{ longtycon} = \text{ty} \rightsquigarrow \text{sig}' : \text{Sig}} \quad (229)$$

Rule 229: Note that type EL type expression ty is evaluated using the ambient scope.

6.6.9 Type Expressions

$$\boxed{\Gamma \vdash \text{ty} \rightsquigarrow \text{con} : \Omega}$$

$$\overline{\Gamma \vdash \text{int} \rightsquigarrow \text{Int} : \Omega} \quad (230)$$

Rule 230: There are analogous rules for the other base types (Float, Bool, Unit, String, and so on), and the base type constructors (ref).

$$\overline{\Gamma \vdash \text{exn} \rightsquigarrow \text{Tagged} : \Omega} \quad (231)$$

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{\text{tyvar}} \rightsquigarrow \text{path} : \Omega}{\Gamma \vdash \text{tyvar} \rightsquigarrow \text{path} : \Omega} \quad (232)$$

$$\frac{\begin{array}{c} \Gamma \vdash \text{ty}_1 \rightsquigarrow \text{con}_1 : \Omega \quad \dots \quad \Gamma \vdash \text{ty}_n \rightsquigarrow \text{con}_n : \Omega \\ \sigma \text{ a permutation of } 1..n \\ \text{lab}_{\sigma(1)} < \dots < \text{lab}_{\sigma(n)} \end{array}}{\Gamma \vdash \{\text{lab}_1 : \text{ty}_1, \dots, \text{lab}_n : \text{ty}_n\} \rightsquigarrow \{\overline{\text{lab}_{\sigma(1)} : \text{con}_{\sigma(1)}}, \dots, \overline{\text{lab}_{\sigma(n)} : \text{con}_{\sigma(n)}}\} : \Omega} \quad (233)$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\text{ctx}} \overline{\text{longtycon}} \rightsquigarrow \text{path} : \Omega^n \Rightarrow \Omega \\ \forall i \in 1..n : \Gamma \vdash \text{ty}_i \rightsquigarrow \text{con}_i : \Omega \end{array}}{\Gamma \vdash (\text{ty}_1, \dots, \text{ty}_n) \text{ longtycon} \rightsquigarrow \text{path} (\text{con}_1, \dots, \text{con}_n) : \Omega} \quad (234)$$

$$\frac{\Gamma \vdash \text{ty} \rightsquigarrow \text{con} : \Omega \quad \Gamma \vdash \text{ty}' \rightsquigarrow \text{con}' : \Omega}{\Gamma \vdash \text{ty} \rightarrow \text{ty}' \rightsquigarrow \text{con} \rightarrow \text{con}' : \Omega} \quad (235)$$

Rule 235: There is no way to express total (\rightarrow) types in the external language.

6.6.10 Type Definitions

$$\boxed{\Gamma \vdash \text{tybind} \rightsquigarrow \text{sbnds} : \text{sdecs}}$$

$$\frac{\begin{array}{c} \text{var}_1, \dots, \text{var}_n \notin \text{BV}(\Gamma) \\ \Gamma, \overline{\text{tyvar}_1} \triangleright \text{var}_1 : \Omega, \dots, \overline{\text{tyvar}_n} \triangleright \text{var}_n : \Omega \vdash \text{ty} \rightsquigarrow \text{con}' : \Omega \\ \text{con} = \lambda(\text{var}_1, \dots, \text{var}_n). \text{con}' \\ \langle \Gamma \vdash \text{tybind} \rightsquigarrow \text{sbnds} : \text{sdecs} \quad \text{var} \notin \text{FV}(\text{sdecs}) \rangle \end{array}}{\Gamma \vdash (\overline{\text{tyvar}_1}, \dots, \overline{\text{tyvar}_n}) \text{ tycon} = \text{ty} \langle \text{and tybind} \rangle \rightsquigarrow \overline{\text{tycon}} \triangleright \text{var} = \text{con} \langle, \text{sbnds} \rangle : \overline{\text{tycon}} \triangleright \text{var} : \Omega^n \Rightarrow \Omega = \text{con} \langle, \text{sdecs} \rangle} \quad (236)$$

6.6.11 Type Descriptions

$$\boxed{\Gamma \vdash \text{typdesc} \rightsquigarrow [\text{sdecs}]}$$

$$\frac{\langle \Gamma \vdash \text{typdesc} \rightsquigarrow \text{sdecs} \rangle}{\Gamma \vdash \text{type} (\overline{\text{tyvar}_1}, \dots, \overline{\text{tyvar}_n}) \text{ id} \langle \text{and typdesc} \rangle \rightsquigarrow \overline{\text{id}} : \Omega^n \Rightarrow \Omega \langle, \text{sdecs} \rangle} \quad (237)$$

$$\frac{\begin{array}{c} \text{var}_1, \dots, \text{var}_n \notin \text{BV}(\Gamma) \\ \Gamma, \overline{\text{tyvar}_1} \triangleright \text{var}_1 : \Omega, \dots, \overline{\text{tyvar}_n} \triangleright \text{var}_n : \Omega \vdash \text{ty} \rightsquigarrow \text{con}' : \Omega \\ \text{con} := \lambda(\text{var}_1, \dots, \text{var}_n). \text{con}' \\ \langle \Gamma \vdash \text{typdesc} \rightsquigarrow \text{sdecs} \quad \text{var} \notin \text{FV}(\text{sdecs}) \rangle \end{array}}{\Gamma \vdash \text{type} (\overline{\text{tyvar}_1}, \dots, \overline{\text{tyvar}_n}) \text{ id} = \text{ty} \langle \langle \text{and typdesc} \rangle \rangle \rightsquigarrow \overline{\text{id}} : \Omega^n \Rightarrow \Omega = \text{con} \langle, \text{sdecs} \rangle} \quad (238)$$

6.6.12 Equality Type Descriptions

$$\boxed{\Gamma \vdash \text{etypdesc} \rightsquigarrow \text{sdecs}}$$

The polymorphic equality functions defined by this judgment the same as the default “structural” equality supplied by SML. Essentially a type is an equality type if and only if we can use this judgment to create the equality function at that type.

$$\frac{\begin{array}{c} \text{sig}' = [\overline{\text{tyvar}_1}^* : [\overline{\text{tyvar}_1} \triangleright \text{var}_1 : \Omega, \text{eq} : \text{var}_1 \times \text{var}_1 \rightarrow \text{Bool}], \dots, \\ \overline{\text{tyvar}_n}^* : [\overline{\text{tyvar}_n} \triangleright \text{var}_n : \Omega, \text{eq} : \text{var}_n \times \text{var}_n \rightarrow \text{Bool}]] \\ \text{con} = \text{var}(\text{var}' . \overline{\text{tyvar}_1}^* . \text{tyvar}_1, \dots, \text{var}' . \overline{\text{tyvar}_n}^* . \text{tyvar}_n) \\ \langle \Gamma \vdash \text{etypdesc} \rightsquigarrow \text{sdecs} \rangle \end{array}}{\Gamma \vdash (\overline{\text{tyvar}_1}, \dots, \overline{\text{tyvar}_n}) \text{ id} \langle \text{and etypdesc} \rangle \rightsquigarrow \overline{\text{id}}^* : [\overline{\text{id}} \triangleright \text{var} : \Omega^n \Rightarrow \Omega, \text{eq} : (\text{var}' : \text{sig}') \rightarrow [\text{it} : \text{con} \times \text{con} \rightarrow \text{Bool}]] \langle, \text{sdecs} \rangle} \quad (239)$$

6.6.13 Datatype Definitions

$$\boxed{\Gamma \vdash \text{datbind} \rightsquigarrow \text{sbnds} : \text{sdecs}}$$

Without loss of generality we may assume that there are no duplicate *tyvar*’s among the bindings.

$$\begin{aligned}\Gamma' &:= \Gamma, \overline{tycon_1} \triangleright var_1^{dt} : \Omega^{n_1} \Rightarrow \Omega \dots \overline{tycon_p} \triangleright var_p^{dt} : \Omega^{n_p} \Rightarrow \Omega \\ \Gamma'' &:= \Gamma', \overline{tyvar_{11}} \triangleright var_{11} : \Omega, \dots, \overline{tyvar_{pn_p}} \triangleright var_{pn_p} : \Omega, \\ &var_1, \dots, var_n \notin BV(\Gamma'')\end{aligned}$$

$$\Gamma'' \vdash \left\{ \begin{array}{c} \text{unit} \\ \text{or} \\ ty_{ij} \end{array} \right\}_{ij} \rightsquigarrow con_{ij} : \Omega$$

$$\begin{aligned}con_i^{\text{sum}} &:= \Sigma(\overline{id_{i1}} \mapsto con_{i1}, \dots, \overline{id_{im_i}} \mapsto con_{im_i}) \\ con'_i &:= \pi_i(\mu \lambda(var_1^{dt}, \dots, var_p^{dt}). ((\lambda(var_{j1}, \dots, var_{jn_j}). con_j^{\text{sum}})_{j=1}^n)) \\ con_i &:= con'_i(var_{i1}, \dots, var_{in_i})\end{aligned}$$

$$\langle \Gamma' \vdash_{\text{eq}} \forall(var_{i1}, \dots, var_{in_i}). con_i \rightsquigarrow \Lambda(var_{i1}^{\langle \text{eq} \rangle_{(i1)'}} , \dots, var_{in_i}^{\langle \text{eq} \rangle_{(in_i)'}}). exp_i^{\text{eq}} \rangle_{i'}$$

$$\begin{aligned}mod_i &:= [\overline{tycon_i} \triangleright var_i^{dt} = con'_i, \\ &\langle \text{eq} = \Lambda(var_{i1}^{\langle \text{eq} \rangle_{(i1)'}} , \dots, var_{in_i}^{\langle \text{eq} \rangle_{(in_i)'}}). exp_i^{\text{eq}} \rangle_{i'} \\ &\overline{id_{ij}} = \Lambda(var_{i1}, \dots, var_{in_i}). \left\{ \begin{array}{c} \text{roll}^{con_i}(\text{inj}_{id_{ij}}^{con_i^{\text{sum}}} \{\}) \\ \text{or} \\ \lambda(var : con_{ij}) : con_i. \text{roll}^{con_i}(\text{inj}_{id_{ij}}^{con_i^{\text{sum}}} var) \end{array} \right\}_{ij}, \quad (\text{for } j \in 1..m_i) \\ &\text{expose} = \Lambda(var_{i1}, \dots, var_{in_i}). \lambda(var : con_i) : con_i^{\text{sum}}. \text{unroll } var] : sig_i\end{aligned}$$

$$\begin{aligned}sig_i &:= [\overline{tycon_i} \triangleright var_i : \Omega^{n_i} \Rightarrow \Omega = var_i^{dt}, \\ &\langle \text{eq} : \forall(var_{i1}^{\langle \text{eq} \rangle_{(i1)'}} , \dots, var_{in_i}^{\langle \text{eq} \rangle_{(in_i)'}}). var_i \times var_i \rightarrow \text{Bool} \rangle_{i'}, \\ &\overline{id_{ij}} : \forall(var_{i1}, \dots, var_{in_i}). \left\{ \begin{array}{c} con_i \\ \text{or} \\ con_{ij} \rightarrow con_i \end{array} \right\}, \quad (\text{for } j \in 1..m_i) \\ &\text{expose} : \forall(var_{i1}, \dots, var_{in_i}). con_i \rightarrow con_i^{\text{sum}}] \\ mod &:= [\overline{tycon_1} \triangleright var_1^{dt} = con'_1, \dots, \overline{tycon_p} \triangleright var_p^{dt} = con'_p, \overline{tycon_1}^* = mod_1, \dots, \overline{tycon_p}^* = mod_p] \\ sig &:= [\overline{tycon_1} \triangleright var_1^{dt} : \Omega^{n_1} \Rightarrow \Omega, \dots, \overline{tycon_p} \triangleright var_p^{dt} : \Omega^{n_p} \Rightarrow \Omega, \overline{tycon_1}^* : sig_1, \dots, \overline{tycon_p}^* : sig_p]\end{aligned}$$

$$\Gamma \vdash (tyvar_{11}, \dots, tyvar_{1n_1}) \quad tycon_1 = id_{11} \left\{ \begin{array}{c} \text{or} \\ \text{of } ty_{11} \end{array} \right\}_{11} | \dots | id_{1m_1} \left\{ \begin{array}{c} \text{or} \\ \text{of } ty_{1m_1} \end{array} \right\}_{1m_1}$$

and ... and

$$(tyvar_{p1}, \dots, tyvar_{pn_p}) \quad tycon_p = id_{p1} \left\{ \begin{array}{c} \text{or} \\ \text{of } ty_{p1} \end{array} \right\}_{p1} | \dots | id_{pm_p} \left\{ \begin{array}{c} \text{or} \\ \text{of } ty_{pm_p} \end{array} \right\}_{pm_p} \rightsquigarrow$$

$$1^* = (mod : sig) : 1^* : sig$$

(240)

6.7 Polymorphic Instantiation

$$\boxed{decs \vdash_{\text{inst}} \rightsquigarrow [sbnds_v] : [sdecs']}$$

$$\frac{\begin{array}{c} decs \vdash con : \Omega \\ \langle decs \vdash_{\text{eq}} con \rightsquigarrow exp_v \rangle \\ \langle \langle decs \vdash_{\text{inst}} \rightsquigarrow [sbnds_v] : [sdecs] \rangle \rangle \end{array}}{decs \vdash_{\text{inst}} \rightsquigarrow [lab' = [lab \triangleright var = con \langle, eq = exp_v \rangle] \langle \langle, sbnds_v \rangle \rangle] : [lab' : [lab \triangleright var : \Omega = con \langle, eq : var \times var \rightarrow \text{Bool} \rangle]] \langle \langle, sdecs \rangle \rangle} \quad (241)$$

Rule 241 Nondeterministically choose types and the corresponding equality functions so as to match a fully-transparent signature.

6.8 Equality Compilation

$$\boxed{decs \vdash_{\text{eq}} con \rightsquigarrow exp}$$

$$\frac{}{decs \vdash_{\text{eq}} \text{Int} \rightsquigarrow \lambda(var_1 : \text{Int}, var_2 : \text{Int}). \text{Bool}(var_1 =_{\text{Int}} var_2)} \quad (242)$$

Rule 242: We assume primitive equality operations already exist at type `Int`, with similar rules for the other base types.

$$\frac{\begin{array}{c} con = \{lab_1 : con_1, \dots, lab_n : con_n\} \\ decs \vdash_{\text{eq}} con_1 \rightsquigarrow exp_1 \quad \dots \quad decs \vdash_{\text{eq}} con_n \rightsquigarrow exp_n \end{array}}{decs \vdash_{\text{eq}} con \rightsquigarrow \lambda(var_1 : con, var_2 : con) : \text{Bool}. exp_1 (\pi_{lab_1} var_1, \pi_{lab_1} var_2) \text{ and } \dots \text{ and } exp_n (\pi_{lab_n} var_1, \pi_{lab_n} var_2)} \quad (243)$$

Rule 243: Records are compared component-wise for equality.

$$\frac{\begin{array}{c} \forall i \in 1..n : \quad con_{\langle lab \rangle} := \Sigma_{\langle lbl \rangle} (con_1, \dots, con_n) \\ \left\{ \begin{array}{l} decs \vdash_{\text{eq}} con_i \rightsquigarrow exp_i \\ exp'_i = \lambda var' : con_{lab_i}. exp_i (\text{proj}_{lab_i}^{con} var', \text{pproj}_{lab_i}^{con} (var_2,)) \end{array} \right. \end{array}}{decs \vdash_{\text{eq}} con \rightsquigarrow \lambda(var_1 : con, var_2 : con) : \text{Bool}. \text{catch case}^{con} var_1 \text{ of } exp'_1, \dots, exp'_n \text{ end with false}} \quad (244)$$

Rule 244: Values of a sum type are equal if they have the same tag and the tagged values are equal.

$$\begin{array}{c}
k \in 1..n \\
\forall i \in 1..n : \quad \begin{cases} con_i = (\pi_i (\mu con')) con'' \\ con'_i := (\pi_i (con' (\mu con'))) con'' \end{cases} \\
decs' = decs, var_1 : [1 \triangleright var : \Omega = con_1, eq : var \times var \rightarrow Bool], \dots, \\
\quad var_n : [1 \triangleright var : \Omega = con_n, eq : var \times var \rightarrow Bool] \\
\forall i \in 1..n : \quad \begin{cases} decs' \vdash_{eq} Unroll con'_i \rightsquigarrow exp_i \\ exp_i^{eq} = (\{var_1^{eq}/var'_1.eq\} \dots \{var_n^{eq}/var'_n.eq\} exp_i) \\ \quad (unroll (\pi_1 var), unroll (\pi_2 var)) \end{cases} \\
\hline
decs \vdash_{eq} con_k \rightsquigarrow \text{fix } (var_i^{eq} (var : con_i \times con_i) : Bool \rightarrow exp_i^{eq})_{i=1}^n \text{ end } var_j^{eq}
\end{array} \quad (245)$$

Rule 245: A recursive type generates recursively-defined equality functions.

$$\frac{decs \vdash path.eq : path.lab \times path.lab \rightarrow Bool}{decs \vdash_{eq} path.lab \rightsquigarrow path.eq} \quad (246)$$

Rule 246: If the constructor is simply a path—the name of an abstract type—we look for an equality function for that type in the same structure.

$$\begin{array}{c}
con = path.lab (con_1, \dots, con_n) \\
decs \vdash_{inst} \rightsquigarrow mod_p : sig_p \\
decs \vdash path.eq : sig_p \rightarrow [it : con \times con \rightarrow Bool] \\
\hline
decs \vdash_{eq} con \rightsquigarrow (path.eq (mod_v)).it
\end{array} \quad (247)$$

Rule 247: Same as Rule 246, except here we deal with the application of an abstract type to a tuple of types. The instantiation judgment may recursively invoke the equality compiler, in order to create equality functions for con_1, \dots, con_n .

$$\frac{decs \vdash con \equiv con' : \Omega \quad decs \vdash_{eq} con' \rightsquigarrow exp}{decs \vdash_{eq} con \rightsquigarrow exp} \quad (248)$$

$$\boxed{decs \vdash_{eq} sig \rightsquigarrow mod}$$

$$\begin{array}{c}
sig = [1^* : [1 \triangleright var_1 : \Omega], \dots, n^* : [n \triangleright var_n : \Omega]] \\
sig' := [1^* : [1 \triangleright var_1 : \Omega \langle eq : var_1 \times var_1 \rightarrow Bool \rangle_1], \dots, n^* : [n \triangleright var_n : \Omega \langle eq : var_n \times var_n \rightarrow Bool \rangle_n]] \\
decs, var : sig' \vdash_{eq} con \rightsquigarrow exp \\
\hline
decs \vdash_{eq} var : sig \rightarrow [it : con] \rightsquigarrow \lambda var : sig'. [it = exp]
\end{array} \quad (249)$$

6.9 Pattern Compilation

Patterns

$$\boxed{\Gamma \vdash pat \Leftarrow exp : con \text{ else } exp' \rightsquigarrow sbnds : sdecs}$$

The judgment should be read as “the bindings of the result of matching exp to the pattern pat are $sbnds$.”

$$\frac{\text{Rules 255}\Gamma\text{257}\Gamma \text{ and 259 do not apply.} \quad \Gamma \vdash exp : con}{\Gamma \vdash id \Leftarrow exp : con \text{ else } exp' \rightsquigarrow \overline{id} = exp : \overline{id} : con} \quad (250)$$

Rule 250: Pattern match against a variable (which is not a datatype or exception constructor).

$$\frac{\begin{array}{c} lab \text{ fresh} \quad type(scon) = con \\ \Gamma \vdash exp : con \end{array}}{\Gamma \vdash scon \Leftarrow exp : con \text{ else } exp' \rightsquigarrow \begin{array}{l} lab = \text{if } exp =_{con} scon \text{ then } \{\} \text{ else raise}^{Unit} exp' : lab : Unit \end{array}} \quad (251)$$

Rule 251: Pattern match against a constant. We need primitive equality functions for constants which can appear in patterns.

$$\frac{lab \text{ fresh} \quad \Gamma \vdash exp : con}{\Gamma \vdash _ \Leftarrow exp : con \text{ else } exp' \rightsquigarrow lab = exp : lab : con} \quad (252)$$

Rule 252: Pattern match against a wildcard.

$$\frac{\begin{array}{c} \Gamma \vdash ty \rightsquigarrow con' : \Omega \quad \Gamma \vdash con \equiv con' : \Omega \\ \Gamma \vdash pat \Leftarrow exp : con \text{ else } exp' \rightsquigarrow sbnds : sdecs \end{array}}{\Gamma \vdash pat : ty \Leftarrow exp : con \text{ else } exp' \rightsquigarrow sbnds : sdecs} \quad (253)$$

Rule 253: Pattern match against an explicitly-typed pattern.

$$\frac{\begin{array}{c} \Gamma \vdash_{ctx} \overline{longid} \rightsquigarrow path.lab_i : con_i \rightarrow con \\ \Gamma \vdash path.expose : con \rightarrow \Sigma(lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n) \\ \Gamma \vdash pat \Leftarrow pproj_{lab_i}^{\Sigma(lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n)} ((path.expose exp), exp') : con_i \text{ else } exp' \rightsquigarrow sbnds : sdecs \end{array}}{\Gamma \vdash longid pat \Leftarrow exp : con \text{ else } exp' \rightsquigarrow sbnds : sdecs} \quad (254)$$

Rule 254: Pattern match against a monomorphic datatype constructor which carries a value.

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path.lab}_i : \text{con} \quad \text{lab fresh} \\
\Gamma \vdash \text{exp} : \text{con} \\
\Gamma \vdash \text{path.expose} : \text{con} \rightarrow \Sigma(\text{lab}_1 \mapsto \text{con}_1, \dots, \text{lab}_n \mapsto \text{con}_n) \\
\hline
\Gamma \vdash \text{longid} \Leftarrow \text{exp} : \text{con} \text{ else } \text{exp}' \rightsquigarrow \\
\text{lab} = \text{pproj}_{\text{lab}_i}^{\Sigma(\text{lab}_1 \mapsto \text{con}_1, \dots, \text{lab}_n \mapsto \text{con}_n)} ((\text{path.expose exp}), \text{exp}') : \text{lab} : \text{Unit}
\end{array} \tag{255}$$

Rule 255: Pattern match against a constant, monomorphic datatype constructor.

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path.lab}_i : \text{sig} \\
\Gamma \vdash \text{path.expose} : \text{sig}_p \rightarrow [\text{it} : \text{con} \rightarrow \Sigma(\text{lab}_1 \mapsto \text{con}_1, \dots, \text{lab}_n \mapsto \text{con}_n)] \\
\Gamma \vdash_{\text{inst}} \rightsquigarrow \text{mod}_p : \text{sig}_p \\
\Gamma \vdash \text{pat} \Leftarrow \text{pproj}_{\text{lab}_i}^{\Sigma(\text{lab}_1 \mapsto \text{con}_1, \dots, \text{lab}_n \mapsto \text{con}_n)} (((\text{path.expose mod}_p).\text{it exp}), \text{exp}') : \text{con}_i \text{ else } \text{exp}' \rightsquigarrow \\
\text{sbnds} : \text{sdecs} \\
\hline
\Gamma \vdash \text{longid pat} \Leftarrow \text{exp} : \text{con} \text{ else } \text{exp}' \rightsquigarrow \text{sbnds} : \text{sdecs}
\end{array} \tag{256}$$

Rule 256: Pattern match against a polymorphic datatype constructor carrying a value.

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path.lab}_i : \text{sig} \\
\Gamma \vdash \text{exp} : \text{con} \\
\Gamma \vdash \text{path.expose} : \text{sig}_p \rightarrow [\text{it} : \text{con} \rightarrow \Sigma(\text{lab}_1 \mapsto \text{con}_1, \dots, \text{lab}_n \mapsto \text{con}_n)] \\
\Gamma \vdash_{\text{inst}} \rightsquigarrow \text{mod}_p : \text{sig}_p \quad \text{lab fresh} \\
\hline
\Gamma \vdash \text{longid} \Leftarrow \text{exp} : \text{con} \text{ else } \text{exp}' \rightsquigarrow \\
\text{lab} = \text{pproj}_{\text{lab}_i}^{\Sigma(\text{lab}_1 \mapsto \text{con}_1, \dots, \text{lab}_n \mapsto \text{con}_n)} ((\text{path.expose mod}_p).\text{it exp}, \text{exp}') : \text{lab} : \text{Unit}
\end{array} \tag{257}$$

Rule 257: Pattern match against a constant polymorphic constructor.

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path.lab} : \text{con} \rightarrow \text{Tagged} \\
\Gamma \vdash \text{path.tag} : \text{con Tag} \\
\Gamma \vdash \text{pat} \Leftarrow (\text{iftagof exp is path.tag then } \lambda \text{var} : \text{con}. \text{var} \text{ else raise}^{\text{con}} \text{exp}') \text{ else } \text{exp}' \rightsquigarrow \\
\text{sbnds} : \text{sdecs} \\
\hline
\Gamma \vdash \text{longid pat} \Leftarrow \text{exp} : \text{con} \text{ else } \text{exp}' \rightsquigarrow \text{sbnds} : \text{sdecs}
\end{array} \tag{258}$$

Rule 258: Pattern match against value-carrying exception constructor.

$$\begin{array}{c}
\Gamma \vdash_{\text{ctx}} \overline{\text{longid}} \rightsquigarrow \text{path.lab} : \text{Tagged} \quad \text{lab fresh} \\
\Gamma \vdash \text{exp} : \text{con} \\
\Gamma \vdash \text{path.tag} : \text{Unit Tag} \\
\hline
\Gamma \vdash \text{longid} \Leftarrow \text{exp} : \text{con} \text{ else } \text{exp}' \rightsquigarrow \\
\text{lab} = \text{iftagof exp is path.tag then } \lambda \text{var} : \text{Unit}. \{\} \text{ else raise}^{\text{Unit}} \text{exp}' : \text{lab} : \text{Unit}
\end{array} \tag{259}$$

Rule 259: Pattern match against constant exception constructor.

$$\begin{array}{c}
\Gamma \vdash con \equiv \{lab'_1:con'_1, \dots, lab'_k:con'_k\} : \Omega \\
\{\overline{lab}_1, \dots, \overline{lab}_n\} \subseteq \{lab'_1, \dots, lab'_n\} \\
\forall i \in 1..n : \Gamma, lab \triangleright var:con \vdash pat_i \Leftarrow \pi_{\overline{lab}_i} exp : con_i \text{ else } exp' \rightsquigarrow sbnds_i : sdecs_i \\
\hline
\Gamma \vdash \{lab_1 = pat_1, \dots, lab_n = pat_n \langle, \dots \rangle\} \Leftarrow exp : con \text{ else } exp' \rightsquigarrow \\
sbnds_1, \dots, sbnds_n : sdecs_1, \dots, sdecs_n
\end{array} \quad (260)$$

Rule 260: Pattern match against a record of patterns. Because we disallow repeated variables in patterns, the syntactic concatenation of structure here is well-formed.

$$\begin{array}{c}
\Gamma \vdash pat_1 \Leftarrow exp : con \text{ else } exp' \rightsquigarrow sbnds_1 : sdecs_1 \\
\Gamma \vdash pat_2 \Leftarrow exp : con \text{ else } exp' \rightsquigarrow sbnds_2 : sdecs_2 \\
\hline
\Gamma \vdash pat_1 \text{ as } pat_2 \Leftarrow exp : con \text{ else } exp' \rightsquigarrow sbnds_1, sbnds_2 : sdecs_1, sdecs_2
\end{array} \quad (261)$$

Rule 261: Pattern match against two patterns simultaneously.

$$\begin{array}{c}
\Gamma \vdash pat \Leftarrow \text{get } exp : con \text{ else } exp' \rightsquigarrow sbnds : sdecs \\
\hline
\Gamma \vdash \text{ref } pat \Leftarrow exp : con \text{ Ref else } exp' \rightsquigarrow sbnds : sdecs
\end{array} \quad (262)$$

Rule 262: Pattern match involving implicit dereferencing of a ref cell.

$$\begin{array}{c}
\Gamma \vdash pat \Leftarrow exp : con' \text{ else } exp' \rightsquigarrow sbnds : sdecs \\
\Gamma \vdash con \equiv con' : \Omega \\
\hline
\Gamma \vdash pat \Leftarrow exp : con \text{ else } exp' \rightsquigarrow sbnds : sdecs
\end{array} \quad (263)$$

6.10 Coercion Compilation

$$\boxed{decs; path: sig_0 \vdash_{\text{sub}} sdec \rightsquigarrow sbnd : sdec'}$$

$$\frac{decs; var_0: sig_0 \vdash_{\text{sig}} lab \rightsquigarrow labs : con'}{decs; path: sig_0 \vdash_{\text{sub}} lab \triangleright var: con \rightsquigarrow lab \triangleright var = var_0.labs : lab \triangleright var: con} \quad (264)$$

Rule 264: Coercion of a monomorphic value specification to a monomorphic value specification.

$$\frac{decs; var_0: sig_0 \vdash_{\text{sig}} lab \rightsquigarrow labs : con \rightarrow con'}{decs; path: sig_0 \vdash_{\text{sub}} lab \triangleright var: con \rightarrow con' \rightsquigarrow lab \triangleright var = \partial var_0.labs : lab \triangleright var: con \rightarrow con'} \quad (265)$$

$$\frac{\begin{array}{l} decs, var'_p: sig'_p; var_0: sig_0 \vdash_{\text{sig}} lab \rightsquigarrow labs : sig_p \rightarrow [it: con] \\ decs, var'_p: sig'_p, var_0: sig_0 \vdash_{\text{inst}} \rightsquigarrow mod_p : sig_p \end{array}}{decs; path: sig_0 \vdash_{\text{sub}} lab \triangleright var: (var'_p: sig'_p) \rightarrow [it: con] \rightsquigarrow \begin{array}{l} lab \triangleright var = \lambda var'_p: sig'_p. (var_0.labs mod_p) : \\ lab \triangleright var: (var'_p: sig'_p) \rightarrow [it: con] \end{array}} \quad (266)$$

Rule 266: Coercion of a polymorphic value specification to a polymorphic value specification; this may involve implicit polymorphic instantiation. (The rule also handles matching polymorphic datatype constructors to `val` specifications.) Note that sig_p and sig'_p need not have the same labels, so this rule also handles alpha-conversion of EL type variables.

$$\frac{\begin{array}{l} decs, var'_p: sig'_p; var_0: sig_0 \vdash_{\text{sig}} lab \rightsquigarrow labs : sig_p \rightarrow [it: con \rightarrow con'] \\ decs, var'_p: sig'_p, var_0: sig_0 \vdash_{\text{inst}} \rightsquigarrow mod_p : sig_p \end{array}}{decs; path: sig_0 \vdash_{\text{sub}} lab \triangleright var: (var'_p: sig'_p) \rightarrow [it: con \rightarrow con'] \rightsquigarrow \begin{array}{l} lab \triangleright var = \lambda var'_p: sig'_p. [it = \partial ((var_0.labs mod_p).it)] : \\ lab \triangleright var: (var'_p: sig'_p) \rightarrow [it: con \rightarrow con'] \end{array}} \quad (267)$$

$$\frac{\begin{array}{l} decs; var_0: sig_0 \vdash_{\text{sig}} lab \rightsquigarrow labs : var_p: sig \rightarrow [it: con] \\ decs \vdash_{\text{inst}} \rightsquigarrow mod : sig \end{array}}{decs; path: sig_0 \vdash_{\text{sub}} lab \triangleright var: con \rightsquigarrow lab \triangleright var = (var_0.labs mod).it : lab \triangleright var: con} \quad (268)$$

Rule 268: Coercion of a polymorphic value (or datatype constructor) to match a monomorphic value specification.

$$\frac{\begin{array}{l} decs; var_0: sig_0 \vdash_{\text{sig}} lab \rightsquigarrow labs : var_p: sig \rightarrow [it: con \rightarrow con'] \\ decs \vdash_{\text{inst}} \rightsquigarrow mod : sig \end{array}}{decs; path: sig_0 \vdash_{\text{sub}} lab \triangleright var: con \rightarrow con' \rightsquigarrow lab \triangleright var = \partial ((var_0.labs mod).it) : lab \triangleright var: con \rightarrow con'} \quad (269)$$

$$\frac{\begin{array}{l} decs; var_0: sig_0 \vdash_{\text{sig}} lab \rightsquigarrow labs : kn d \\ \langle decs, var_0: sig_0 \vdash var_0.labs \equiv con : kn d \rangle \end{array}}{decs; path: sig_0 \vdash_{\text{sub}} lab \triangleright var: kn d (=con) \rightsquigarrow lab \triangleright var = var_0.labs : lab \triangleright var: kn d = var_0.labs} \quad (270)$$

Rule 270: Coercion of a type component to a type specification.

$$\begin{array}{c}
\text{decs}; \text{var}_0 : \text{sig}_0 \vdash_{\text{sig}} \text{lab} \rightsquigarrow \text{labs} : \Omega \\
\langle \text{decs}, \text{var}_0 : \text{sig}_0 \vdash \text{var}_0.\text{labs} \equiv \text{con} : \Omega \rangle \\
\text{decs}, \text{var}_0 : \text{sig}_0 \vdash_{\text{eq}} \text{var}_0.\text{labs} \rightsquigarrow \text{exp} \\
\hline
\text{decs}; \text{path} : \text{sig}_0 \vdash_{\text{sub}} \text{lab}^* : [\text{lab} \triangleright \text{var} : \Omega \langle = \text{con} \rangle, \text{eq} : \text{var} \times \text{var} \rightarrow \text{Bool}] \rightsquigarrow \\
\text{lab}'^* : [\text{lab} \triangleright \text{var} = \text{var}_0.\text{labs}, \text{eq} : \text{exp}] : \\
\text{lab}'^* : [\text{lab} \triangleright \text{var} : \Omega = \text{var}_0.\text{labs}, \text{eq} : \text{var} \times \text{var} \rightarrow \text{Bool}]
\end{array} \quad (271)$$

Coercion of a type declaration corresponding to an eqtype signature specification involving no type variables. We invoke the equality compiler to create an equality function for the type being copied.

$$\begin{array}{c}
\text{decs}; \text{var}_0 : \text{sig}_0 \vdash_{\text{sig}} \text{lab} \rightsquigarrow \text{labs} : \Omega^n \Rightarrow \Omega \\
\text{con}' = \text{var}(\text{var}_p.\text{lab}'_1.\text{lab}_1, \dots, \text{var}_p.\text{lab}'_n.\text{lab}_n) \\
\text{decs}, \text{var}_0 : \text{sig}_0, \text{var}_p : \text{sig}_p \vdash_{\text{eq}} \text{con}' \rightsquigarrow \text{exp} \\
\hline
\text{decs}; \text{path} : \text{sig}_0 \vdash_{\text{sub}} \text{lab}^* : [\text{lab} \triangleright \text{var} : \Omega^n \Rightarrow \Omega, \text{eq} : \text{var}_p : \text{sig}_p \rightarrow [\text{it} : \text{con}' \times \text{con}' \rightarrow \text{Bool}]] \rightsquigarrow \\
\text{lab}^* : [\text{lab} \triangleright \text{var} = \text{var}_0.\text{labs}, \text{eq} : \lambda \text{var}_p : \text{sig}_p. [\text{it} = \text{exp}]] : \\
\text{lab}^* : [\text{lab} \triangleright \text{var} : \Omega^n \Rightarrow \Omega \langle = \text{var}_0.\text{labs} \rangle, \text{eq} : \text{var}_p : \text{sig}_p \rightarrow [\text{it} : \text{con}' \times \text{con}' \rightarrow \text{Bool}]]
\end{array} \quad (272)$$

Coercion of a type declaration correspond to a eqtype signature specification involving type variables (i.e., eqtype ('a,'b) foo).

$$\begin{array}{c}
\text{decs}; \text{var}_0 : \text{sig}_0 \vdash_{\text{sig}} \text{lab} \rightsquigarrow \text{labs} : \text{sig}' \\
\text{decs}, \text{var}_0 : \text{sig}_0 \vdash_{\text{sub}} \text{path}.\text{labs} : \text{sig}' \preceq \text{sig} \rightsquigarrow \text{mod}_c : \text{sig}_c \\
\hline
\text{decs}; \text{path} : \text{sig}_0 \vdash_{\text{sub}} \text{lab} \triangleright \text{var} : \text{sig} \rightsquigarrow \text{lab} \triangleright \text{var} = \text{mod}_c : \text{lab} \triangleright \text{var} : \text{sig}_c
\end{array} \quad (273)$$

Rule 273: Coercion of a module component.

$$\boxed{\text{decs} \vdash_{\text{sub}} \text{path} : \text{sig}_0 \preceq \text{sig} \rightsquigarrow \text{mod} : \text{sig}'}$$

$$\begin{array}{c}
\text{decs}; \text{path} : \text{sig}_0 \vdash_{\text{sub}} \text{lab}_1 \triangleright \text{dec}_1 \rightsquigarrow \text{sbnd}_1 : \text{sdec}_1 \\
\text{decs}, \text{dec}_1; \text{path} : \text{sig}_0 \vdash_{\text{sub}} \text{lab}_2 \triangleright \text{dec}_2 \rightsquigarrow \text{sbnd}_2 : \text{sdec}_2 \\
\vdots \\
\text{decs}, \text{dec}_1, \dots, \text{dec}_{n-1}; \text{path} : \text{sig}_0 \vdash_{\text{sub}} \text{lab}_n \triangleright \text{dec}_n \rightsquigarrow \text{sbnd}_n : \text{sdec}_n \\
\hline
\text{decs} \vdash_{\text{sub}} \text{path} : \text{sig}_0 \preceq [\text{lab}_1 \triangleright \text{dec}_1, \dots, \text{lab}_n \triangleright \text{dec}_n] \rightsquigarrow \\
[\text{sbnd}_1, \dots, \text{sbnd}_n] : [\text{sdec}_1, \dots, \text{sdec}_n]
\end{array} \quad (274)$$

$$\begin{array}{c}
\text{decs} \vdash_{\text{sub}} \text{var}_2 : \text{sig}_2 \preceq \text{sig}_1 \rightsquigarrow \text{mod}_3 : \text{sig}_3 \\
\text{decs}, \text{var}_2 : \text{sig}_2 \vdash_{\text{sub}} \text{var}'_1 : \text{sig}'_1 \preceq \text{sig}'_2 \rightsquigarrow \text{mod}_4 : \text{sig}_4 \\
\text{decs}, \text{var}_2 : \text{sig}_2 \vdash \text{mod}_4 : \text{sig}_4 \\
\hline
\text{decs} \vdash_{\text{sub}} \text{path} : (\text{var}_1 : \text{sig}_1 \rightarrow \text{sig}'_1) \preceq (\text{var}_2 : \text{sig}_2 \rightarrow \text{sig}'_2) \rightsquigarrow \\
\lambda \text{var}_2 : \text{sig}_2. \text{let } \text{var}'_1 = \text{path } \text{mod}_3 \text{ in } \text{mod}_4 \text{ end} : \\
\text{var}_2 : \text{sig}_2 \rightarrow \text{sig}_4
\end{array} \quad (275)$$

Rule 275: Coercions for functor subtyping. We also need the same rule for subtyping of total functors (not shown here).

6.11 Signature Patching

where type

$$\boxed{sig \vdash_{wt} labs := con : kno \rightsquigarrow sig' : Sig}$$

This judgment should be read “By adding to the signature sig the fact that the abstract type component selected by $labs$ is equal to $con : kno$, we get the signature sig' .”

$$\frac{\begin{array}{l} FV(con) \cap BV(sdec's) = \emptyset \\ sig = [sdec's, lab \triangleright var : kno, sdec's'] \end{array}}{sig \vdash_{wt} lab := con : kno \rightsquigarrow [sdec's, lab \triangleright var : kno = con, sdec's'] : Sig} \quad (276)$$

$$\frac{\begin{array}{l} FV(con) \cap BV(sdec's) = \emptyset \\ sig = [sdec's, lab \triangleright var : sig', sdec's'] \\ sig' \vdash_{wt} labs := con : kno \rightsquigarrow sig'' : Sig \end{array}}{sig \vdash_{wt} lab.labs := con : kno \rightsquigarrow [sdec's, lab \triangleright var : sig'', sdec's'] : Sig} \quad (277)$$

sharing

$$\boxed{sig \vdash_{sh} labs_1 := labs_2 : kno \rightsquigarrow sig' : Sig}$$

This judgment should be read “By adding to the signature sig the fact that the (abstract) type components of kind kno selected by $labs_1$ and $labs_2$ are equal, we get the signature sig' .”

$$\frac{sig = [sdec's, lab' \triangleright var' : kno, sdec's', lab \triangleright var : kno, sdec'']}{sig \vdash_{sh} lab := lab' : kno \rightsquigarrow [sdec's, lab' \triangleright var' : kno, sdec's', lab \triangleright var : kno = var', sdec''] : Sig} \quad (278)$$

$$\frac{\begin{array}{l} sig = [sdec's, lab' \triangleright var' : kno, sdec's', lab \triangleright var : sig, sdec''] \\ sig \vdash_{wt} labs := var' : kno \rightsquigarrow sig' : Sig \end{array}}{sig \vdash_{sh} lab.labs := lab' : kno \rightsquigarrow [sdec's, lab' \triangleright var' : kno, sdec's', lab \triangleright var : sig', sdec''] : Sig} \quad (279)$$

$$\frac{\begin{array}{l} sig = [sdec's, lab' \triangleright var' : sig', sdec's', lab \triangleright var : sig'', sdec''] \\ sig'' \vdash_{wt} labs := var'.labs' : kno \rightsquigarrow sig''' : Sig \end{array}}{sig \vdash_{sh} lab.labs := lab'.labs' : kno \rightsquigarrow [sdec's, lab' \triangleright var' : sig', sdec's', lab \triangleright var : sig''', sdec''] : Sig} \quad (280)$$

$$\frac{\begin{array}{l} sig = [sdec's, lab \triangleright var : sig', sdec's'] \\ sig' \vdash_{sh} labs := labs' : kno \rightsquigarrow sig'' : Sig \end{array}}{sig \vdash_{sh} lab.labs := lab.labs' : kno \rightsquigarrow [sdec's, lab \triangleright var : sig'', sdec's'] : Sig} \quad (281)$$

6.12 Lookup Rules

The lookup rules specify the order in which translation contexts and IL signatures are searched.

- To prevent an explosion of rules the metavariable *class* is used to denote a type or kind as appropriate.
- Any IL structure label starred with an asterisk (e.g. Γlab^*) is treated specially by the lookup which checks inside the structure. That is when looking for an identifier in a context we also look *inside* starred structure declarations. If the identifier is found inside such a structure the full path to the identifier through the open structure is returned.
- Any type or signature returned by a lookup will be valid with respect to the ambient context.

6.12.1 Context Lookup

$$\boxed{\Gamma \vdash_{\text{ctx}} labs \rightsquigarrow path : class}$$

Main rules for looking up identifiers in a translation context.

$$\frac{\Gamma \vdash_{\text{ctx}} labs \rightsquigarrow path \quad \Gamma \vdash path : con}{\Gamma \vdash_{\text{ctx}} labs \rightsquigarrow path : con} \quad (282)$$

$$\frac{\Gamma \vdash_{\text{ctx}} labs \rightsquigarrow path \quad \Gamma \vdash path : kn}{\Gamma \vdash_{\text{ctx}} labs \rightsquigarrow path : kn} \quad (283)$$

$$\frac{\Gamma \vdash_{\text{ctx}} labs \rightsquigarrow path \quad \Gamma \vdash path : sig}{\Gamma \vdash_{\text{ctx}} labs \rightsquigarrow path : sig} \quad (284)$$

$$\boxed{\Gamma \vdash_{\text{ctx}} labs \rightsquigarrow path}$$

“Utility” rules used to look up identifiers in a translation context.

$$\frac{lab = lab'}{\Gamma, lab \triangleright var : con \vdash_{\text{ctx}} lab' \rightsquigarrow var} \quad (285)$$

$$\frac{lab \neq lab' \quad \Gamma \vdash_{\text{ctx}} lab' \rightsquigarrow path}{\Gamma, lab \triangleright var : con \vdash_{\text{ctx}} lab' \rightsquigarrow path} \quad (286)$$

$$\frac{lab = lab'}{\Gamma, lab \triangleright var : kn \langle = con \rangle \vdash_{\text{ctx}} lab' \rightsquigarrow var} \quad (287)$$

$$\frac{lab \neq lab' \quad \Gamma \vdash_{\text{ctx}} lab' \rightsquigarrow path}{\Gamma, lab \triangleright var : kn \langle = con \rangle \vdash_{\text{ctx}} lab' \rightsquigarrow path} \quad (288)$$

$$\frac{lab = lab'}{\Gamma, lab \triangleright var: sig \vdash_{ctx} lab' \rightsquigarrow var} \quad (289)$$

$$\frac{lab \neq lab' \quad \Gamma \vdash_{ctx} lab' \rightsquigarrow path}{\Gamma, lab \triangleright var: sig \vdash_{ctx} lab' \rightsquigarrow path} \quad (290)$$

$$\frac{sig \vdash_{sig} lab' \rightsquigarrow path}{\Gamma, lab^* \triangleright var: sig \vdash_{ctx} lab' \rightsquigarrow var.path} \quad (291)$$

$$\frac{sig \vdash_{sig} lab' \not\rightsquigarrow \quad \Gamma \vdash_{ctx} lab' \rightsquigarrow path}{\Gamma, lab^* \triangleright var: sig \vdash_{ctx} lab' \rightsquigarrow path} \quad (292)$$

$$\frac{\Gamma \vdash_{ctx} lab \rightsquigarrow path : sig \quad \Gamma; path: sig \vdash_{sig} labs \rightsquigarrow labs' : class}{\Gamma \vdash_{ctx} lab.labs \rightsquigarrow path.labs'} \quad (293)$$

6.12.2 Signature Lookup

$decs; path: sig \vdash_{sig} labs \rightsquigarrow labs' : class$

Main rules for looking up identifiers in a signature.

$$\frac{decs \vdash sig : Sig \quad sig \vdash_{sig} lab \rightsquigarrow labs' \quad decs \vdash path.labs' : con}{decs; path: sig \vdash_{sig} lab \rightsquigarrow labs' : con} \quad (294)$$

$$\frac{decs \vdash sig : Sig \quad sig \vdash_{sig} lab \rightsquigarrow labs' \quad decs \vdash path.labs' : sig'}{decs; path: sig \vdash_{sig} lab \rightsquigarrow labs' : sig'} \quad (295)$$

$$\frac{decs \vdash sig : Sig \quad sig \vdash_{sig} lab \rightsquigarrow labs' \quad decs \vdash path.labs' : kno}{decs; path: sig \vdash_{sig} lab \rightsquigarrow labs' : kno} \quad (296)$$

$$\frac{\begin{array}{c} decs \vdash sig : Sig \\ decs; path: sig \vdash_{sig} lab \rightsquigarrow labs' : sig' \\ decs; path.labs': sig' \vdash_{sig} labs \rightsquigarrow labs'' : class \end{array}}{decs; path: sig \vdash_{sig} lab.labs \rightsquigarrow labs'.labs'' : class} \quad (297)$$

$sig \vdash_{sig} lab \rightsquigarrow labs'$

“Utility” rules used to look up identifiers in a signature.

$$\frac{lab = lab'}{[sdecs, lab \triangleright var: con] \vdash_{sig} lab' \rightsquigarrow lab} \quad (298)$$

$$\frac{lab \neq lab' \quad [sdecs] \vdash_{sig} lab' \rightsquigarrow labs}{[sdecs, lab \triangleright var: con] \vdash_{sig} lab' \rightsquigarrow labs} \quad (299)$$

$$\frac{lab = lab'}{[sdecs, lab \triangleright var:knd(=con)] \vdash_{sig} lab' \rightsquigarrow lab} \quad (300)$$

$$\frac{lab \neq lab' \quad [sdecs] \vdash_{sig} lab' \rightsquigarrow labs}{[sdecs, lab \triangleright var:knd(=con)] \vdash_{sig} lab' \rightsquigarrow labs} \quad (301)$$

$$\frac{lab = lab'}{[sdecs, lab \triangleright var:sig] \vdash_{sig} lab' \rightsquigarrow lab} \quad (302)$$

$$\frac{lab \neq lab' \quad [sdecs] \vdash_{sig} lab' \rightsquigarrow labs}{[sdecs, lab \triangleright var:sig] \vdash_{sig} lab' \rightsquigarrow labs} \quad (303)$$

$$\frac{sig \vdash_{sig} lab' \rightsquigarrow labs}{[sdecs, lab^* \triangleright var:sig] \vdash_{sig} lab' \rightsquigarrow lab^*.labs} \quad (304)$$

$$\frac{sig \vdash_{sig} lab' \not\rightsquigarrow \quad [sdecs] \vdash_{sig} lab' \rightsquigarrow labs}{[sdecs, lab^* \triangleright var:sig] \vdash_{sig} lab' \rightsquigarrow labs} \quad (305)$$

6.13 Overloading

For each overloaded identifier we wish to add to the EL we add one translation rule for each overloaded type. For example we may want to overload the EL identifier $+$ for addition at types `int` and `real`. In this case we would add two rules along the lines of:

$$\overline{\Gamma \vdash + \rightsquigarrow +_{int} : int \times int \rightarrow int}$$

and

$$\overline{\Gamma \vdash + \rightsquigarrow +_{float} : float \times float \rightarrow float},$$

where $+_{int}$ and $+_{float}$ are the two IL primitive addition functions.

7 Properties

The proofs presented here are only sketches; the Standard ML language is far too large to check every case by hand in a reasonable amount of time. Future work may involve attempting to check these proofs with an automated system.

7.1 Properties of the Internal Language

7.1.1 Static Semantics

Lemma 1 (Well-formedness)

The following propositions hold:

1. If $decs \vdash dec \text{ ok}$ then $\vdash decs \text{ ok}$.

2. If $\text{decs} \vdash \text{bnd} : \text{dec}$ then $\text{decs} \vdash \text{dec} \text{ ok}$.
3. If $\text{decs} \vdash \text{knd} : \text{Kind}$ then $\vdash \text{decs} \text{ ok}$.
4. If $\text{decs} \vdash \text{con} : \text{knd}$ then $\text{decs} \vdash \text{knd} : \text{Kind}$.
5. If $\text{decs} \vdash \text{con} \equiv \text{con}' : \text{knd}$ then $\text{decs} \vdash \text{con} : \text{knd}$ and $\text{decs} \vdash \text{con}' : \text{knd}$.
6. If $\text{decs} \vdash \text{exp} : \text{con}$ then $\text{decs} \vdash \text{con} : \Omega$.
7. If $\text{decs} \vdash \text{sdecs} \text{ ok}$ then $\vdash \text{decs} \text{ ok}$.
8. If $\text{decs} \vdash \text{sig} : \text{Sig}$ then $\vdash \text{decs} \text{ ok}$.
9. If $\text{decs} \vdash \text{sdecs} \leq \text{sdecs}'$ then $\text{decs} \vdash \text{sdecs} \text{ ok}$ and $\text{decs} \vdash \text{sdecs}' \text{ ok}$.
10. If $\text{decs} \vdash \text{sig} \leq \text{sig}' : \text{Sig}$ then $\text{decs} \vdash \text{sig} : \text{Sig}$ and $\text{decs} \vdash \text{sig}' : \text{Sig}$.
11. If $\text{decs} \vdash \text{sdecs} \equiv \text{sdecs}'$ then $\text{decs} \vdash \text{sdecs} \text{ ok}$ and $\text{decs} \vdash \text{sdecs}' \text{ ok}$.
12. If $\text{decs} \vdash \text{sig} \equiv \text{sig}' : \text{Sig}$ then $\text{decs} \vdash \text{sig} : \text{Sig}$ and $\text{decs} \vdash \text{sig}' : \text{Sig}$.
13. If $\text{decs} \vdash \text{sbnds} : \text{sdecs}$ then $\text{decs} \vdash \text{sdecs} \text{ ok}$.
14. If $\text{decs} \vdash \text{mod} : \text{sig}$ then $\text{decs} \vdash \text{sig} : \text{Sig}$.
15. If $\text{decs} \vdash \text{exp} \downarrow \text{con}$ then $\text{decs} \vdash \text{exp} : \text{con}$. If in addition $\text{decs} \vdash \text{exp} : \text{con}'$, then $\text{decs} \vdash \text{exp} \downarrow \text{con}'$.
16. If $\text{decs} \vdash \text{mod} \downarrow \text{sig}$ then $\text{decs} \vdash \text{mod} : \text{sig}$. If in addition $\text{decs} \vdash \text{mod} : \text{sig}'$, then $\text{decs} \vdash \text{mod} \downarrow \text{sig}'$.

The following lemma states that internal language judgments are preserved under substitution of values for free variables in a typing judgment Γ where a value is defined syntactically in Figure 4 to be a phrase in evaluated form.

Proposition 2 (Substitution)

1. If $\text{decs}, \text{var} : \text{class}, \text{decs}' \vdash \text{con}_1 \equiv \text{con}_2 : \text{knd}$ and $\text{decs} \vdash \text{val} : \text{class}$ then $\text{decs}, \{\text{val}/\text{var}\} \text{decs}' \vdash \{\text{val}/\text{var}\} \text{con}_1 \equiv \{\text{val}/\text{var}\} \text{con}_2 : \text{knd}$.
2. If $\text{decs}, \text{var} : \text{class}, \text{decs}' \vdash \text{sig}_1 \equiv \text{sig}_2 : \text{Sig}$ and $\text{decs} \vdash \text{val} : \text{class}$ then $\text{decs}, \{\text{val}/\text{var}\} \text{decs}' \vdash \{\text{val}/\text{var}\} \text{sig}_1 \equiv \{\text{val}/\text{var}\} \text{sig}_2 : \text{Sig}$.
3. If $\text{decs}, \text{var} : \text{class}, \text{decs}' \vdash \text{sig}_1 \leq \text{sig}_2 : \text{Sig}$ and $\text{decs} \vdash \text{val} : \text{class}$ then $\text{decs}, \{\text{val}/\text{var}\} \text{decs}' \vdash \{\text{val}/\text{var}\} \text{sig}_1 \leq \{\text{val}/\text{var}\} \text{sig}_2 : \text{Sig}$.
4. If $\text{decs}, \text{var} : \text{class}, \text{decs}' \vdash \text{exp} : \text{con}$ and $\text{decs} \vdash \text{val} : \text{class}$ then $\text{decs}, \{\text{val}/\text{var}\} \text{decs}' \vdash \{\text{val}/\text{var}\} \text{exp} : \{\text{val}/\text{var}\} \text{con}$.
5. If $\text{decs}, \text{var} : \text{class}, \text{decs}' \vdash \text{mod} : \text{sig}$ and $\text{decs} \vdash \text{val} : \text{class}$ then $\text{decs}, \{\text{val}/\text{var}\} \text{decs}' \vdash \{\text{val}/\text{var}\} \text{mod} : \{\text{val}/\text{var}\} \text{sig}$.

6. If $decs, var:class, decs' \vdash con : kn_d$ and $decs \vdash val : sig'$ then $decs, \{val/var\} decs' \vdash \{val/var\} con : kn_d$.
7. If $decs, var:class, decs' \vdash exp \downarrow con$ and $decs \vdash val : class$ then $decs, \{val/var\} decs' \vdash \{val/var\} exp \downarrow \{val/var\} con$.
8. If $decs, var:class, decs' \vdash mod \downarrow sig$ and $decs \vdash val : class$ then $decs, \{val/var\} decs' \vdash \{val/var\} mod \downarrow \{val/var\} sig$.

Proof (sketch): [By induction on the derivation of the first premises; due to space constraints we show only two sample cases]

- Case: The derivation ends with Typing Rule 49:

$$\frac{\begin{array}{l} decs, var:class, decs' \vdash exp : con_2 \rightarrow con \\ decs, var:class, decs' \vdash exp' : con_2 \end{array}}{decs, var:class, decs' \vdash exp \ exp' : con}$$

and $decs \vdash val : class$. By the inductive hypotheses Γ

$$\begin{array}{l} decs, \{val/var\} decs' \vdash \{val/var\} exp : \\ (\{val/var\} con_2) \rightarrow (\{val/var\} con) \end{array}$$

and

$$decs, \{val/var\} decs' \vdash \{val/var\} exp' : \{val/var\} con_2.$$

Therefore from Typing Rule 49 Γ we can conclude

$$decs, \{val/var\} decs' \vdash \{exp_v/var\} (exp \ exp') : con.$$

- Case: the derivation ends with Typing Rule 31:

$$\frac{\begin{array}{l} decs, var:class, decs' \vdash mod_v : [lab_1:dec_1, \dots, lab_m:dec_m, \\ lab:var:kn_d=con_2, sdecs] \\ decs, var:class, decs', dec_1, \dots, dec_m \vdash con \equiv con_2 : kn_d \\ decs, var:class, decs' \vdash con : kn_d \end{array}}{decs, var:class, decs' \vdash mod_v.lab \equiv con : kn_d}$$

and $decs \vdash val : class$. By the using properties 5 Γ 1 Γ and 6 of the induction hypothesis Γ and Typing Rule 31 Γ we have

$$decs, \{val/var\} decs' \vdash \{val/var\} mod_v.lab \equiv \{val/var\} con : kn_d.$$

■

Claim 3 (Decidability)

All internal language judgments considered by the elaborator are decidable.

Proof (sketch): Because all internal-language module expressions generated by the elaborator have most-specific signatures, the decidability of the static semantics can be reduced in a straightforward fashion to the decidability of constructor equivalence. Of the constructor equivalence judgments, Rules 32–45 describe a familiar simply-typed lambda calculus with records and constants. Even allowing the use of simple abbreviations in the context (Rule 30) can be seen to preserve decidability. Roughly speaking one can always first “inline” the abbreviations and then test for equivalence as in the previous case.

It is not obvious how to extend the proof to include Rule 31 in a simple fashion. Lilibridge [Lil97] gives a complex argument for the decidability of constructor equivalence in a related system (without records of constructors). We conjecture an analogous argument would apply for our system. ■

7.1.2 Dynamic Semantics

We define two states to be *equivalent* written

$$(\Delta, \sigma, E, \text{phrase}) \cong (\Delta', \sigma', E', \text{phrase}'),$$

if the two states are equal componentwise up to consistent renaming of the locations and exception tags appearing in Δ and normal renaming of bound-variables. This is clearly an equivalence relation on states which preserves the property of being a terminal state.

Lemma 4 (Determinacy of Evaluation)

The following properties hold:

1. If Σ is terminal and $\Sigma \cong \Sigma'$, then Σ' is also terminal.
2. If $\Sigma \hookrightarrow \Sigma_1$ and $\Sigma \cong \Sigma'$, then there exists a state $\Sigma'_1 \cong \Sigma_1$ such that $\Sigma' \hookrightarrow \Sigma'_1$.
3. If $\Sigma_1 \hookrightarrow \Sigma'_1$, $\Sigma_2 \hookrightarrow \Sigma'_2$ and $\Sigma_1 \cong \Sigma_2$ then $\Sigma'_1 \cong \Sigma'_2$.
4. If $\Sigma \hookrightarrow^* \Sigma_t$ and $\Sigma \hookrightarrow^* \Sigma'_t$ then $\Sigma_t \cong \Sigma'_t$.

A context C is a phrase with a single *hole* written \square . We write $C[\text{phrase}]$ for the result of filling the hole in C with *phrase*, possibly incurring variable capture. A stack of frames E determines a context \widehat{E} in the obvious way: $\widehat{\square}$ is the context \square and $\widehat{E \circ F}$ is the context $\widehat{E}[F]$.

Proposition 5 (Decomposition & Replacement)

1. If $\text{decs} \vdash \widehat{E}[\text{exp}] : \text{con}$ and exp is closed, then $\text{decs} \vdash \text{exp} : \text{con}'$ for some type con' . Furthermore, if $\text{decs} \vdash \text{exp}' : \text{con}'$ where exp' is closed, then $\text{decs} \vdash \widehat{E}[\text{exp}'] : \text{con}$.
2. If $\text{decs} \vdash \widehat{E}[\text{mod}] : \text{con}$ and mod is closed, then $\text{decs} \vdash \text{mod} : \text{sig}$ for some signature sig . Furthermore, if $\text{decs} \vdash \text{mod}' : \text{sig}$ where mod' is closed, then $\text{decs} \vdash \widehat{E}[\text{mod}'] : \text{con}$.
3. If $\text{decs} \vdash \widehat{E}[\text{con}'] : \text{con}$ and con' is closed, then $\text{decs} \vdash \text{con} : \text{knd}$ for some kind knd . Furthermore, if $\text{decs} \vdash \text{con}'' : \text{knd}$ where con'' is closed, then $\text{decs} \vdash \widehat{E}[\text{con}''] : \text{con}$.

Proof (sketch): [By induction on E]

If $E = []$ then the proposition is trivial since $\widehat{E}[\text{phrase}] = \text{phrase}$. Hence assume $E = E' \circ F$. Due to space constraints we show just one typical case.

- Case: $F = [] \text{ exp}_2$. Then $\widehat{E}[\text{exp}] = \widehat{E'}[\text{exp exp}_2]$. By the inductive hypothesis we have $\text{decs} \vdash \text{exp exp}_2 : \text{con}'$ for some con' . Inspection of the typing rules shows that this derivation must contain an application of Rule 49 or Rule 50; we show here only the former case the other is similar. By inversion $\text{decs} \vdash \text{exp} : \text{con}_2 \rightarrow \text{con}'$ and $\text{decs} \vdash \text{exp}_2 : \text{con}_2$ for some type con_2 . Furthermore if $\text{decs} \vdash \text{exp}' : \text{con}_2 \rightarrow \text{con}'$ then $\text{decs} \vdash \text{exp}' \text{ exp} : \text{con}'$ and by the inductive hypothesis we have $\text{decs} \vdash \widehat{E'}[\text{exp}' \text{ exp}_2] : \text{con}$; that is $\text{decs} \vdash \widehat{E}[\text{exp}] : \text{con}$. ■

7.1.3 Soundness of the Internal Language

Following Harper [Har93] we say that a store σ is well-formed with respect to a context Δ written $\Delta \vdash \sigma$ if

$$\forall \text{loc} \in \text{BV}(\Delta), \text{ if } \Delta \vdash \text{loc} : \text{con Ref} \text{ then } \Delta \vdash \sigma(\text{loc}) : \text{con}.$$

This formulation of store typing avoids the need for complex maximal fixed point constructions [Tof90]. (An essentially similar observation was made by Wright and Felleisen [WF91].)

Fix a base type ans of *answers* to which a complete closed program might evaluate.² We can say that a machine state is well-formed written

$$\vdash (\Delta, \sigma, E, \text{phrase}),$$

if and only if $\Delta \vdash \widehat{E}[\text{phrase}] : \text{ans}$ phrase is closed and $\Delta \vdash \sigma$. An important property of the internal language semantics is that well-formedness of a state is preserved by evaluation.

Proposition 6 (Preservation)

If $\vdash (\Delta, \sigma, E, \text{phrase})$ and

$$(\Delta, \sigma, E, \text{phrase}) \hookrightarrow (\Delta', \sigma', E', \text{phrase}')$$

then $\vdash (\Delta', \sigma', E', \text{phrase}')$.

Proof (sketch):

If the transition is a search rule then preservation follows trivially for $\Delta = \Delta' \Gamma \sigma = \sigma' \Gamma$ and $\widehat{E}[\text{phrase}] = \widehat{E'}[\text{phrase}']$. Hence we need only consider the reduction rules. We show only one sample case here.

²A reasonable choice might be *String*, or *Unit* if we model all I/O by updating the store; the particular choice does not affect our results.

- Case: Transition Rule 160:

$$(\Delta, \sigma, E \circ (\pi_{lab} []), \{rbnds_v, lab=exp_v, rbnds_v'\}) \hookrightarrow (\Delta, \sigma, E, exp_v)$$

Then $E \circ (\pi_{lab} [])[\{rbnds_v, lab=exp_v, rbnds_v'\}] = \widehat{E}[\pi_{lab} \{rbnds_v, lab=exp_v, rbnds_v'\}]$.

By the well-formedness assumption and Decomposition Γ

$decs \vdash \pi_{lab} \{rbnds_v, lab=exp_v, rbnds_v'\} : con'$ for some type con' . Since this judgement must be proven using Typing Rules 54 and 53 Γ by inversion we have that

$decs \vdash exp_v : con'$. Since exp_v is must by closed by the well-formedness assumption and $decs \vdash \widehat{E}[exp_v] : ans$ by Replacement Γ well-formedness is preserved. ■

Furthermore Γ evaluation of a well-typed program can never “get stuck”: if a well-formed state is not terminal Γ there is always an applicable transition to another (well-formed) state. The proof relies on a characterization of the shapes of closed values of each type.

Lemma 7 (Canonical Forms)

1. Assume $\Delta \vdash exp_v' : con'$ where exp_v' is closed.

<i>If con' is of the form...</i>	<i>then exp_v' is of the form...</i>
$con_1 \rightarrow con_2$	$\pi_{\bar{k}} \text{ fix fbnds end}$
$con_1 \rightarrow con_2$	$\pi_{\bar{l}} \text{ fix fbnd end}$
$\{lab_1 : con_1, \dots, lab_n : con_n\}$	$\left\{ \begin{array}{l} \{lab_1 = exp_{v_1}, \dots, lab_n = exp_{v_n}\} \\ \text{fix fbnds end} \end{array} \right.$
$\Sigma_{\langle lab_i \rangle} (lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n)$	$\text{inj}_{lab_i}^{\Sigma(lab_1 \mapsto con_1, \dots, lab_n \mapsto con_n)} exp_v$
$(\pi_i (\mu con)) con'$	$\text{roll}^{(\pi_i (\mu con)) con'} exp_v$
Tagged	$\text{tag}(tag, exp_v)$
con Ref	loc
base type	$scon$

2. Assume $\Delta \vdash mod_v' : sig'$ where mod_v' is closed.

<i>If sig' is of the form...</i>	<i>then mod_v' is of the form...</i>
$[sdecs]$	$[sbnds_v]$
$var : sig \rightarrow sig'$	$\lambda var : sig. mod$
$var : sig \rightarrow sig'$	$\lambda var : sig. mod$

Proof: By inspection of the applicable typing rules for values. For example Γ consider the case in which $\Delta \vdash exp_v' : con'$ where con' is a record type. There are numerous rules that allow the conclusion that an expression has a record type (application Γ record projection Γ etc.) but only one—the rule for record expressions—applies in the case of a closed value. This is the only possible form for exp_v . The other cases are analogous. ■

Proposition 8 (Progress)

If $\vdash \Sigma$ then either Σ is terminal or there exists a state Σ' such that $\Sigma \hookrightarrow \Sigma'$.

Proof (sketch):

There are three cases:

1. Σ is terminal.
2. $\Sigma = (\Delta, \sigma, \mu, E, \text{phrase})$ where *phrase* is not a value. By inspection for every possible phrase syntax there is an applicable rule in the dynamic semantics (a reduction rule for `new_tag[con]` and search rules otherwise).
3. $\Sigma = (\Delta, \sigma, \mu, E \circ F, \text{val})$. This has one subcase for each possible frame F . We show one such case:
 - Case: $F = (\text{get } [])$. By well-formedness and Decomposition $\Gamma \Delta \vdash \text{get } \text{val} : \text{con}'$ for some type con' . By inversion of Rule 59 we have that $\Delta \vdash \text{val} : \text{con}'$ Ref. By Canonical Forms Γval is a location and by well-formedness $\text{val} \in \text{BV}(\sigma)$. Therefore Transition Rule 163 applies.

■

7.2 Properties of the Elaborator

The minimal requirement for the elaborator is that the elaboration of EL code yields well-formed IL code:

Proposition 9 (Well-formed translation)

Assume $\vdash \Gamma$ ok. Then the following propositions hold:

1. If $\Gamma \vdash \text{expr} \rightsquigarrow \text{exp} : \text{con}$ then $\Gamma \vdash \text{exp} : \text{con}$.
2. If $\Gamma \vdash \text{match} \rightsquigarrow \text{exp} : \text{con}$ then $\Gamma \vdash \text{exp} : \text{con}$.
3. If $\Gamma \vdash \text{strdec} \rightsquigarrow \text{sbnds} : \text{sdec}$ then $\Gamma \vdash \text{sbnds} : \text{sdec}$.
4. If $\Gamma \vdash \text{strexp} \rightsquigarrow \text{mod} : \text{sig}$ then $\Gamma \vdash \text{mod} : \text{sig}$.
5. If $\Gamma \vdash \text{strbind} \rightsquigarrow \text{sbnds} : \text{sdec}$ then $\Gamma \vdash \text{sbnds} : \text{sdec}$.
6. If $\Gamma \vdash \text{funbind} \rightsquigarrow \text{sbnds} : \text{sdec}$ then $\Gamma \vdash \text{sbnds} : \text{sdec}$.
7. If $\Gamma \vdash \text{spec} \rightsquigarrow \text{sdec}$ then $\Gamma \vdash \text{sdec}$ ok.
8. If $\Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig} : \text{Sig}$ then $\Gamma \vdash \text{sig} : \text{Sig}$.
9. If $\Gamma \vdash \text{ty} \rightsquigarrow \text{con} : \Omega$ then $\Gamma \vdash \text{con} : \Omega$.
10. If $\Gamma \vdash \text{tybind} \rightsquigarrow \text{sbnds} : \text{sdec}$ then $\Gamma \vdash \text{sbnds} : \text{sdec}$.

11. If $\Gamma \vdash \text{typdesc} \rightsquigarrow \text{sbnds} : \text{sdecs}$ then $\Gamma \vdash \text{sbnds} : \text{sdecs}$.
12. If $\Gamma \vdash \text{etypdesc} \rightsquigarrow \text{sbnds} : \text{sdecs}$ then $\Gamma \vdash \text{sbnds} : \text{sdecs}$.
13. If $\Gamma \vdash \text{datbind} \rightsquigarrow \text{sbnds} : \text{sdecs}$ then $\Gamma \vdash \text{sbnds} : \text{sdecs}$.
14. If $\text{decs} \vdash_{\text{inst}} \rightsquigarrow \text{mod} : \text{sig}$ then $\text{decs} \vdash \text{mod} : \text{sig}$. Furthermore, mod is a syntactic value and sig is the most-specific signature of mod .
15. If $\text{decs} \vdash_{\text{eq}} \text{con} \rightsquigarrow \text{exp}$ then $\text{decs} \vdash \text{exp} : \text{con} \times \text{con} \rightarrow \text{Bool}$. Furthermore, exp is a syntactic value. If $\text{decs} \vdash_{\text{eq}} \Lambda(\text{var}_1, \dots, \text{var}_n). \text{con} \rightsquigarrow \text{mod}$ then $\text{decs} \vdash \text{mod} : \forall \text{var}^{(\text{eq})_1}, \dots, \text{var}^{(\text{eq})_n}. \text{con} \times \text{con} \rightarrow \text{Bool}$.
16. If $\Gamma \vdash \text{pat} \Leftarrow \text{exp} : \text{con}$ else $\text{sbnds} \rightsquigarrow \text{sdecs}$; $\Gamma \vdash \text{exp} : \text{Tagged}$ and $\Gamma \vdash \text{exp} : \text{con}$, then $\Gamma \vdash \text{sbnds} : \text{sdecs}$.
17. If $\text{decs}; \text{path} : \text{sig}_0 \vdash_{\text{sub}} \text{sdec} \rightsquigarrow \text{sbind} : \text{sdec}'$ then $\text{decs} \vdash \text{path} : \text{sig}_0$, $\text{decs} \vdash \text{sdec} \text{ ok}$, and $\text{decs} \vdash \text{sbind} : \text{sdec}'$.
18. If $\text{decs} \vdash_{\text{sub}} \text{path} : \text{sig}_0 \preceq \text{sig} \rightsquigarrow \text{mod} : \text{sig}'$ then $\text{decs} \vdash \text{path} : \text{sig}_0$, $\text{decs} \vdash \text{sig}_0 \leq \text{sig} : \text{Sig}$, $\text{decs} \vdash \text{mod} : \text{sig}'$, and $\text{decs} \vdash \text{sig}' \leq \text{sig} : \text{Sig}$.
19. If $\Gamma \vdash_{\text{ctx}} \text{labs} \rightsquigarrow \text{path} : \text{class}$ then $\Gamma \vdash \text{path} : \text{class}$. Furthermore, $\Gamma \vdash \text{path} : \text{class}'$ if and only if $\Gamma \vdash_{\text{ctx}} \text{labs} \rightsquigarrow \text{path} : \text{class}$.
20. If $\Gamma; \text{path} : \text{sig} \vdash_{\text{sig}} \text{labs} \rightsquigarrow \text{labs} : \text{class}$ then $\Gamma \vdash \text{path.labs} : \text{class}$. Furthermore, if $\Gamma \vdash \text{path.labs} : \text{class}'$ then $\Gamma; \text{path} : \text{sig} \vdash_{\text{sig}} \text{labs} \rightsquigarrow \text{labs} : \text{class}'$.
21. If $\text{sig} \vdash_{\text{wt}} \text{labs} := \text{con} : \text{knd} \rightsquigarrow \text{sig}' : \text{Sig}$, $\Gamma \vdash \text{sig} : \text{Sig}$, and $\Gamma \vdash \text{con} : \text{knd}$, then $\Gamma \vdash \text{sig}' : \text{Sig}$.
22. If $\text{sig} \vdash_{\text{sh}} \text{labs} := \text{labs}' : \text{knd} \rightsquigarrow \text{sig}' : \text{Sig}$ and $\Gamma \vdash \text{sig} : \text{Sig}$ then $\Gamma \vdash \text{sig}' : \text{Sig}$.

Proof: This follows by induction on the derivation of the first premises. ■

To claim that we have *defined* a language Γ we need elaboration to be a “function” in the sense that all possible elaborations of an EL program yield “equivalent” IL programs. We argue that such coherence is plausible Γ also we have not attempted a formal proof.

Claim 10 (Coherence)

If

$$\text{basis}^* \triangleright \text{basis} : \text{sig}_{\text{basis}} \vdash \text{expr} \rightsquigarrow \text{exp} : \text{ans}$$

and

$$\text{basis}^* \triangleright \text{basis} : \text{sig}_{\text{basis}} \vdash \text{expr} \rightsquigarrow \text{exp}' : \text{ans}$$

and

$$(\cdot, \cdot, \text{let basis} = \text{mod}_{\text{basis}} \text{ in exp end}) \hookrightarrow^* \Sigma_t$$

then for some terminal state $\Sigma'_t \cong \Sigma_t$,

$$(\cdot, \cdot, \text{let basis} = \text{mod}_{\text{basis}} \text{ in exp}' \text{ end}) \hookrightarrow^* \Sigma'_t.$$

Proof (sketch): We enumerate the instances of non-determinism in the elaborator and argue that these should not cause incoherence.

1. The elaborator may “guess” types for bound variables (in `fn` expressions) and when instantiating polymorphism. Types in “dead” code may be underconstrained as in

`(fn x => 3)(fn y => y)`

where there are infinitely many choices for the type of the identity `fn y => y`. However, the dynamic semantics is sufficiently abstract that choices of types do not affect the course of evaluation.

2. Similarly, the elaborator may have a choice regarding how to resolve overloading. The simplest argument is that each overloaded operator corresponds to a single operator in the dynamic semantics, so that again evaluation does not depend on the resolved type. (A more sophisticated argument should be able to show that in a complete program, whenever there is a choice of how to resolve overloading, the operator will never be applied.)
3. The elaborator must decide whether valuable expressions should be monomorphic or polymorphic, and in the latter case, on how many type variables to generalize. This turns expressions into functor abstractions, which in general causes incoherence by changing the order of evaluation—an expression wrapped by a functor will not be evaluated until the functor is applied. However, we only generalize valuable expressions, which have no effects, so that these choices will not affect observable behavior.
4. There is nondeterminism in the creation of equality functions: there may be different definitions of equality at the same type. However, we maintain the invariant that equality functions always perform structural equality on the underlying values regardless of abstraction. Any two candidate definitions then have the same behavior, and no incoherence can arise.

■

8 Acknowledgments

We would like to thank Perry Cheng and Martin Elsman for their particularly helpful comments on the work presented here.

References

- [Har93] Robert Harper. A simplified account of polymorphic references. Technical Report CMU-CS-93-169, School of Computer Science, Carnegie Mellon University, 1993.

- [HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st Symposium on Principles of Programming Languages*Γpages 123–137Γ1994.
- [HM93] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*Γ15(2):211–252Γ1993.
- [HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Principles of Programming Languages*Γpages 130–141Γ1995.
- [HMM90] Robert HarperΓJohn C. MitchellΓand Eugenio Moggi. Higher-order modules and the phase distinction. In *17th Symposium on Principles of Programming Languages*Γpages 341–354Γ1990.
- [Ler94] Xavier Leroy. Manifest typesΓmodulesΓand separate compilation. In *21st Symposium on Principles of Programming Languages*Γpages 109–122Γ1994.
- [Ler96] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*Γ6(5):667–698Γ1996.
- [Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesisΓSchool of Computer ScienceΓCarnegie Mellon UniversityΓ1997. Available as CMU Technical Report CMU-CS-97-122.
- [Mor95] Greg Morrisett. *Compiling with Types*. PhD thesisΓSchool of Computer ScienceΓCarnegie Mellon UniversityΓ1995. Available as CMU Technical Report CMU-CS-95-226.
- [MT94] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *European Symposium on Programming*Γpages 409–423. Springer-VerlagΓLNCS 788ΓApril 1994.
- [MTH90] Robin MilnerΓMads TofteΓand Robert Harper. *The Definition of Standard ML*. MIT PressΓ1990.
- [MTHM97] Robin MilnerΓMads TofteΓRobert HarperΓand Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT PressΓ1997.
- [Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19ΓAarhus UniversityΓSeptember 1981.
- [Tar96] David Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. PhD thesisΓSchool of Computer ScienceΓCarnegie Mellon UniversityΓ1996.

- [TMC⁺96] David TarditiΓGreg MorrisettΓPerry ChengΓChris StoneΓRobert HarperΓand Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*Γpages 181–192Γ1996.
- [Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*Γ89(1):1–34ΓNovember 1990.
- [WF91] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160ΓDepartment of Computer ScienceΓRice UniversityΓ1991.